SISTEMAS INFORMÁTICOS 2009/2010

Diseño y optimización para ultra-bajo consumo de arquitecturas y aplicaciones de procesado de señal para las redes de sensores inalámbricas

Autores:

Pablo Acevedo Montserrat
Luis Alfonso González de la Calzada
Jorge Guirado Alonso

Directores:

David Atienza Alonso
Pablo García del Valle

Índice

1. Agradecimentos

En primer lugar querríamos dar las gracias a David Atienza Alonso por habernos dado la posibilidad de realizar este proyecto.

También queremos darle las gracias a Pablo García del Valle por hacer mucho más fácil el camino de comprender por qué las herramientas de Xilinx hacen las cosas que hacen.

Tampoco podemos olvidarnos de Amir Hossein Mamaghanian ya que sin su ayuda hubiese sido imposible desarrollar este proyecto.

Y finalmente queríamos dar las gracias Nadia Khaled por esforzarse en hablar en castellano con nosotros y por corregir los errores de nuestra inexperiencia.

2. Resumen / Abstract

En este proyecto se ha desarrollado en hardware el algoritmo FISTA de resolución inversa de problemas lineales en tratamiento de la señal, en su versión constant stepsize. Este diseño reconstruye, a partir de una señal comprimida (con un ratio de compresión de un 50%) mediante el método Compressive Sensing, la señal original de una dimensión. En nuestro caso, la señal será un electrocardiograma (ECG).

La motivación del proyecto es conseguir reducir el consumo de dispositivos portátiles, llamados Shimmer, que muestrean continuamente ECG, comprimiendo las señales mediante el método Compressive Sensing, que es un algoritmo muy simple y muy eficiente en energía consumida, para después reconstruir estas señales en un dispositivo mucho mas potente, ya que el Shimmer no tiene la potencia necesaria para hacerlo de un modo eficiente en tiempo y energía. De esta forma, el Shimmer puede estar muestreando durante días de forma ininterrumpida, y las señales comprimidas pueden ser reconstruidas de una forma rápida y eficiente utilizando este hardware específico. Las aplicaciones de este proyecto son muy variadas, desde control medico en casa, sin la supervisión continua de un medico en el hospital, al entrenamiento de atletas.

In this project, we developed a hardware version of FISTA algorithm, used to solve linear inverse problems, in its constant stepsize variant. This design reconstructs, from a compressed signal (with a 50% compression ratio) using Compressive Sensing, the original one-dimension signal. In this particular case, this signal is an electrocardiogram (ECG).

The aim of this project is to reduce the power consumption of portable devices, named Shimmer nodes, which continuously sample ECG signals from a human body, using Compressive Sensing, which is a very simple and energy efficient algorithm, so that the compressed signal can be uncompressed in a much more powerful device, as the Shimmer lacks throughput to do this quickly and efficiently. This way, the battery on the portable device can last for days, and the compressed signals can be reconstructed very quickly by using this specific hardware. The possible applications of this system vary from medical control at home, without having medical surveillance at hospital, to athletes training.

3. Palabras clave

FPGA, FISTA, electrocardiograma, VHDL, ISE, Xilinx, Compressed sensing, Compressive sampling, Virtex, EDK.

4. Base teórica

4.1. CS

Las aproximaciones convencionales para muestreo de señales siguen el conocido teorema de Shannon: la tasa de muestreo debe ser al menos el doble de la máxima frecuencia presente en la señal (frecuencia de Nyquist). De hecho, este principio funciona en prácticamente todos los protocolos de adquisición de señal en cuanto a sistemas audiovisuales, aparatos médicos etc. se refiere.

Este proyecto de Sistemas Informáticos está basado en algo completamente distinto, un nuevo paradigma que no tiene nada que ver con la tradicional forma de adquirir una señal comentada en el párrafo anterior. Estamos hablando del llamado "**Compressive sampling**" o "**Compressed sensing**" (de ahora en adelante **CS**).

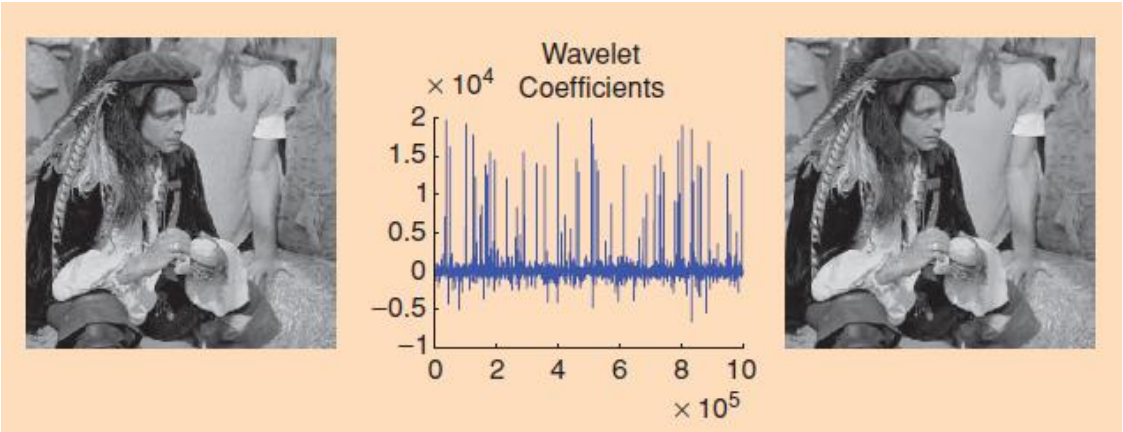
El funcionamiento de esta técnica se basa en dos principios, la **dispersión** (sparsity) que pertenece a la señal en la que estamos interesados y la **incoherencia** (incoherence) que pertenece a la *modalidad de senseo (sensing modality)*

La **dispersión** se basa en la idea de que la "frecuencia de la información" de una señal contínua en el tiempo podría ser mucho más pequeña de la sugerida por su ancho de banda, sin por ello dejar de ser representativa de la misma.

La **incoherencia** dice, que al contrario que la señal en la que estamos interesados, las *sampling/sensing waveforms* tienen una representaicón de datos muy densa.

De todo esto se deduce que se pueden diseñar protocolos de muestreo de señales que capturen información útil contenida en una señal muy dispersa y guardar todo ello en una pequeña cantidad de datos. En otras palabras, CS es un protocolo de adquisición de datos muy simple y eficiente, que muestrea a una baja frecuencia y más tarde usa el poder de computación para reconstruir una señal a partir de lo que aparenta ser una incompleta colección de muestras.

En definitiva, esta teoría concluye que **se pueden recuperar ciertas señales (e imágenes) con bastantes menos muestras que los métodos tradicionales**, tal como se ilustra en la siguiente imagen:



4.2. FISTA

FISTA (**F**ast **I**terative **S**hrinkage **T**hresholding **A**lgorithm for Linear Inverse Problems) es el algoritmo que usamos en el desarrollo de nuestro proyecto. Este algoritmo está basado en la teoría de **CS** expuesta en el apartado anterior. Hay varias versiones del algoritmo, pero nos centraremos en comentar la que hemos usado nosotros, concretamente la *algo aquí*, haciendo una breve referencia a las demás.

FISTA es además es una versión mejorada de un algoritmo anterior llamado ISTA, en el sentido de que converge más rápido al resultado final. Se ha comprobado de que es significativamente mejor, tanto teóricamente como en la práctica.

La versión más básica para resolver problemas de nuestro tipo es "ISTA with constant stepsize":

ISTA with constant stepsize
Input: $L := L(f)$ - A Lipschitz constant of ∇f.
Step 0. Take $\mathbf{x}_0 \in \mathbb{R}^n$.
Step k. $(k \geq 1)$ Compute

$$\mathbf{x}_k = p_L(\mathbf{x}_{k-1}). \tag{3.1}$$

El problema de este algoritmo es que L (cosntante de Lipschitz) no es siempre conocida o computable. Para paliar esta carencia se propone otro el cual detallamos a continuación:

Esta versión no es otra que "ISTA with backtracking":

ISTA with backtracking
Step 0. Take $L_0 > 0$, some $\eta > 1$, and $\mathbf{x}_0 \in \mathbb{R}^n$.
Step k. $(k \geq 1)$ Find the smallest nonnegative integers i_k such that with $\bar{L} = \eta^{i_k} L_{k-1}$

$$F(p_{\bar{L}}(\mathbf{x}_{k-1})) \leq Q_{\bar{L}}(p_{\bar{L}}(\mathbf{x}_{k-1}), \mathbf{x}_{k-1}). \tag{3.2}$$

Set $L_k = \eta^{i_k} L_{k-1}$ and compute

$$\mathbf{x}_k = p_{L_k}(\mathbf{x}_{k-1}). \tag{3.3}$$

Sin embargo aún posible encontrar un algoritmo mejor, que es precisamente el que usaremos en este proyecto de Sistemas Informáticos:

"FISTA with constant stepsize":

FISTA with constant stepsize
Input: $L = L(f)$ - A Lipschitz constant of ∇f.
Step 0. Take $\mathbf{y}_1 = \mathbf{x}_0 \in \mathbb{R}^n$, $t_1 = 1$.
Step k. $(k \geq 1)$ Compute

$$\mathbf{x}_k = p_L(\mathbf{y}_k), \tag{4.1}$$

$$t_{k+1} = \frac{1 + \sqrt{1 + 4t_k^2}}{2}, \tag{4.2}$$

$$\mathbf{y}_{k+1} = \mathbf{x}_k + \left(\frac{t_k - 1}{t_{k+1}}\right)(\mathbf{x}_k - \mathbf{x}_{k-1}). \tag{4.3}$$

Se puede hacer claramente una analogía con nuestro algoritmo (con detalle en el código completo en soporte digital):

(4.1) correspondería a la inicialización de los datos, que en nuestro código son una serie de llamadas a **A, At y proxL1**

(4.2) corresponde a **t=(1+sqrt(1+4*told*told))/2**

(4.3) corresponde al bucle que actualiza **w**

5. Visión global del proyecto

5.1. Idea inicial

La idea inicial es diseñar e implementar un sistema empotrado con un procesador que ejecute el código de descompresión de una señal de electrocardiograma codificada mediante el método FISTA.

El objetivo principal es conseguir una **descompresión lo más cercana al tiempo real** que sea posible. Para ello contamos con una base de datos de muestras de electrocardiograma de una duración de 2 segundos cada una. Por tanto, el objetivo será descomprimir dichas muestras en un tiempo inferior al indicado. Debido a esto, consideramos que una **implementación hardware de algunas de las funciones más costosas** del código inicial nos sería de mucha utilidad, por lo que el sistema final, estará compuesto de un procesador y de módulos hardware, diseñados mediante VHDL, que realizarán la descompresión en paralelo de los datos.

Este sistema recibirá los datos por el puerto serie, utilizando el estándar RS232. Los datos recibidos se almacenarán en memorias de datos dentro del sistema y serán accedidos por el procesador y los distintos módulos hardware.

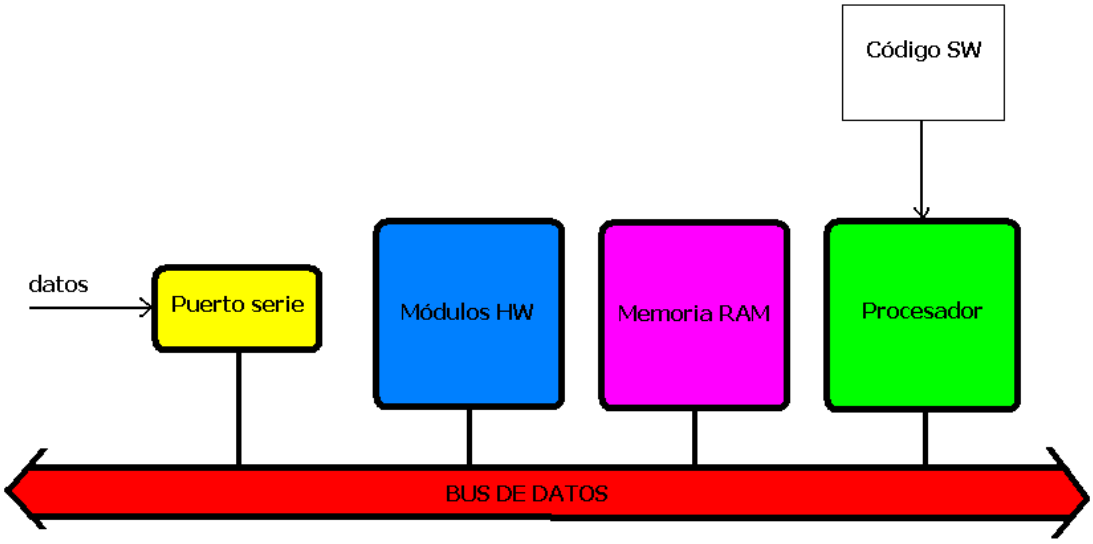


Ilustración 1 - Idea inicial del sistema

5.2. Herramientas utilizadas

En la realización de este proyecto se han utilizado tanto herramientas y entornos software como sistemas hardware.

Hardware:

FPGA Xilinx University Program XUPV5-LX110T

- Xilinx Virtex-5 XC5VLX110T FPGA.
- Dos Xilinx XCF32P Platform Flash PROMs (32 Mbyte cada una) para almacenar grandes configuraciones del dispositivo.
- Xilinx SystemACE Compact Flash configuration controller.
- Un módulo de RAM de 256Mbyte DDR2 DIMM (SODIMM) de 64 bits de ancho compatible con los IP y controladores de software de EDK.
- SRAM síncrona de 32-bit ZBT e Intel P30 StrataFlash.
- Clavija Ethernet 10/100/1000 que soporta MII, GMII, RGMII, y SGMII.
- Host USB y controladores de periféricos.
- Generador de reloj del sistema programable.
- Códec AC97 estéreo con línea de entrada, salida, cascos, micrófono, y jacks SPDIF de audio digital.
- Puerto serie RS-232, display LCD de 16x2 caracteres, y muchos otros dispositivos de entrada/salida.

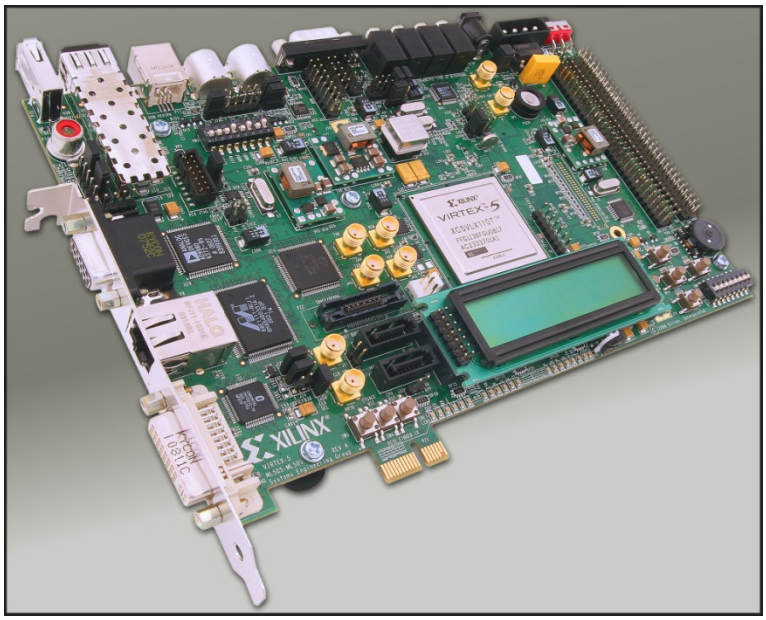


Imagen de la placa utilizada

Software:

Xilinx es una compañía líder en tecnología hardware basada en FPGAs. Adicionalmente, Xilinx proporciona distintos paquetes de herramientas de diseño para la gama de FPGAs que fabrica y comercializa. Dichas herramientas ofrecen un amplio abanico de posibilidades de optimización del diseño. En este proyecto, utilizamos las herramientas de síntesis e implementación que se integran en el paquete **Xilinx ISE Design Suite 12: Embedded Edition.**

De todos los programas ofrecidos por esta suite, nos centramos en tres campos de trabajo bien diferenciados, para los cuales, disponemos de entornos bien definidos:

- ISE Project Navigator
- ISE Simulator (ISim)
- Embedded Development Kit (EDK), que incluye:
  - Xilinx Platform Studio (XPS)
  - Source Development Kit (SDK)

ISE Project Navigator

ISE Project Navigator integra las herramientas que necesitamos para desarrollar sistemas destinados a placas Xilinx y nos permite comenzar a diseñar nuestros propios **módulos hardware en lenguaje VHDL** de forma sencilla con una interfaz gráfica fácil de usar. Proporciona gestión de fuentes de diseño, acceso fácil para ejecutar todos los pasos destinados a la síntesis del diseño, y el acceso al análisis de los resultados del diseño. Además, disponemos de asistentes intuitivos para la introducción de módulos del IP catalog y sugerencias y plantillas de diseño.

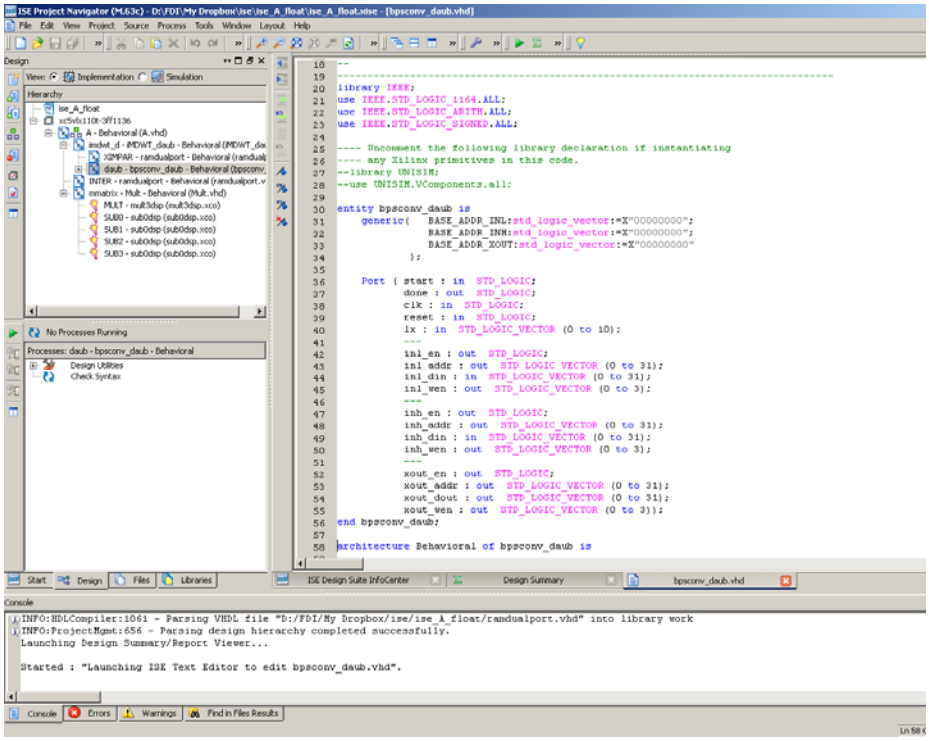


Ilustración 2 - Interfaz del ISE Project Navigator

ISE Simulator (ISim)

Esta herramienta es la destinada a verificar en una primera fase nuestros módulos hardware. Se integra de forma casi transparente con el Project Navigator y nos facilita el testeo mediante una buena interfaz gráfica y multitud de opciones. Para realizar estas simulaciones, implementamos una serie de bancos de pruebas en vhdl, que el propio programa se encarga de interpretar conjuntamente con el diseño.

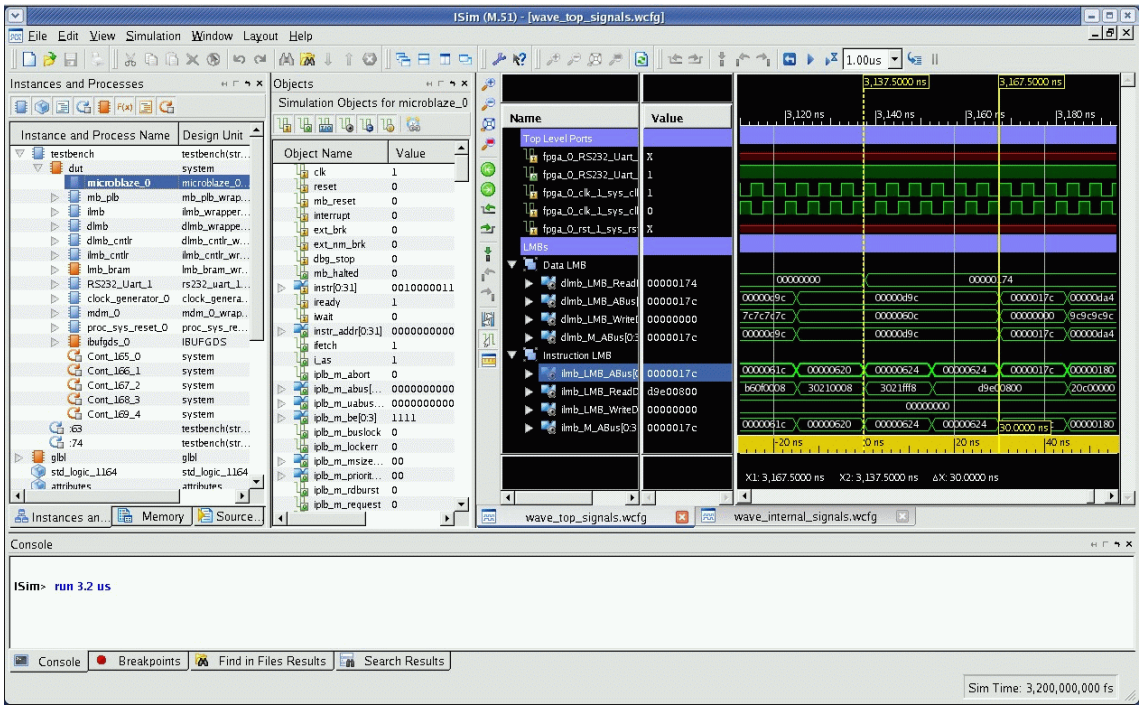


Ilustración 3 - Interfaz de ISim

Embedded Development Kit (EDK)

- *Xilinx Platform Studio:*

Aplicación de desarrollo e integración que permite crear **diseños hardware de sistemas on-chip con procesadores empotrados.** Permite la utilización de asistentes (Base System Builder) para generar rápidamente sistemas completos para uno o varios procesadores. Tiene acceso a una gran cantidad de periféricos destinados a este tipo de plataformas, como controladores de memoria, UART para comunicación por serie…

Está encargado de generar los ficheros de implementación y un archivo de especificación hardware de extensión MHS (*Microprocessor Hardware Specification*) donde está definida la arquitectura del sistema.

LibGen es la herramienta encargada de configurar las bibliotecas y drivers de los elementos creados en el sistema empotrado basándose en el archivo de especificación software MSS (*Microprocessor Software Specification*). A través de la creación de dicho archivo, se especifican los dispositivos estándar de entrada/salida del sistema, las rutinas de atención a las interrupciones y otras características relacionadas con el software. También se incluye una herramienta GNU para compilar, enlazar y ensamblar (mb-gcc).

Posteriormente genera un archivo .bit, que contiene el código necesario para cargar en la placa y que ésta se comporte acuerdo a la implementación realizada.

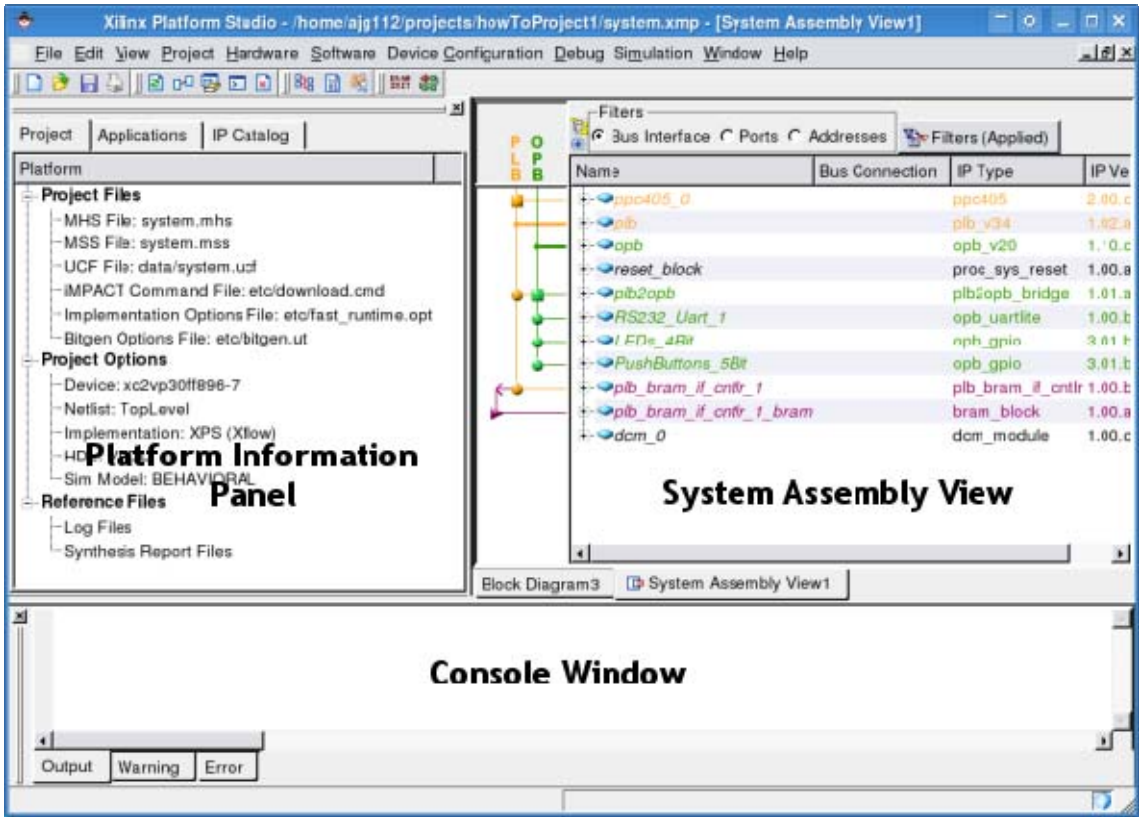


Ilustración 4 - Interfaz de XPS

- *Software Development Kit:*

Mediante la importación desde XPS del diseño hardware, se pueden desarrollar aplicaciones en un entorno más destinado a ello, ya que está basado en Eclipse.

También permite la depuración en tiempo real de estas aplicaciones, mientras se están ejecutando en el hardware empotrado, así como realizar análisis de tiempos y costes de forma sencilla.

5.3. Flujo de trabajo

Estos son los pasos seguidos para desarrollar el trabajo del proyecto:

Fase 1. Obtención de código en C:

Se trata de obtener un código en C funcional que se encargue de descomprimir una señal codificada utilizando el algoritmo FISTA.

Fase 2. Optimización del código en C:

Optimizar el código obtenido al máximo, para simplificar en la medida de lo posible la implementación en hardware.

Fase 3. Verificación del código en C:

En esta fase se obtienen los resultados del código y se estudian los resultados para comprobar su viabilidad.

Fase 4. Estudio del uso de las funciones del código (profiling):

En esta fase se obtienen mediciones de tiempo de las funciones del código y se identifican los puntos críticos en los que se ha de mejorar el tiempo de ejecución. Las funciones críticas obtenidas serán las implementadas en hardware.

Fase 5. Diseño e implementación de módulos hardware:

Se diseñan y se implementan las distintas funciones del código que han sido previamente seleccionadas. Cada miembro del grupo se encarga de unas funciones determinadas, trabajando en paralelo.

Fase 6. Verificación de módulos por separado:

Se testea cada módulo por separado junto con el código original, comparando los resultados obtenidos en software con los obtenidos en hardware.

Fase 7. Integración de todos los módulos hardware en un único periférico:

Se diseña un periférico global con todos los módulos, que será el que se conecte finalmente al bus.

Fase 8. Verificación final de la implementación:

Se comprueban los datos obtenidos mediante la ejecución completa del sistema

6. Implementación SW en detalle

6.1. Optimización de la matriz R

Desde la primera versión del código a la última, la matriz R ha sufrido una modificación considerable.

En lugar de guardar todos los valores –ya que sólo eran 0s y 1s- hemos guardado las posiciones de estos en la matriz original. Cada columna tenía 12 valores no nulos, y precisamente aprovechamos eso para crear la nueva matriz de filas de 12 elementos, lo que nos permite distinguir las dimensiones de la matriz ahorrando espacio.

Esta mejora implica otra. La multiplicación por R se hace significativamente más fácil:

```
double R[]={0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0
           ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0
           ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0
           ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0
           ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0
           ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0
           ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0
           ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0
           ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0
           ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0
           ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0
           ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0
           ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0
           ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0
           ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0
           ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0
           ,0 ,0 ,1 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0
           ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0
           ,0 ,1 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0
           ,0 ,0 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0
           ,0 ,0 ,0 ,0 ,0 ,0 . . . . . . . . . . . }
```

Matriz original

```
int R[]={2,38,53,63,81,87,106,112,118,152,193,207,
         6,69,70,82,97,106,108,126,149,151,193,218,
         8,11,21,60,136,156,167,188,190,199,229,230,
         57,63,73,98,101,109,145,159,170,191,209,224,
         13,32,39,68,100,105,111,113,165,224,228,247,
         0,13,19,36,39,73,114,140,163,169,194,251,
         15,36,66,80,107,167,171,196,210,228,243,244,
         18,33,44,94,103,155,160,182,208,215,217,237,
         4,37,38,90,142,171,180,188,193,197,231,239,
         5,8,23,33,55,79,108,122,153,160,174,254,
         24,35,39,58,61,76,109,134,179,203,205,244,
         86,87,103,109,113,137,186,204,230,247,251,252,
         42,55,77,97,111,126,136,216,239,242,247,249,
         29,36,41,44,58,94,96,97,125,130,167,236,
         55,70,91,97,107,109,133,139,172,201,228,247,
         5,6,8,41,76,79,92,116,176,201,217,226,
         7,24,37,73,113,115,119,182,184,188,220,229,
         35,44,54,60,68,148,191,200,211,220,234,252,
         20,33,43,65,71,140,166,174,181,201,215,247,
         35,45,56,76,114,140,160,175,181,184,208,248,
         6,8,35,61,67,102,118,135,163,185,196,206,
         1,10,21,47,60,62,137,153,185,215,223,250,
         63,65,72,74,100,138,182,194,197,200,234,251,. . . . }
```

Matriz optimizada

También hay que destacar que esta matriz está destinada a una descompresión con un ratio de 0.50, por lo que tanto el software como el hardware están optimizados para estas dimensiones.

Finalmente, el bucle de multiplicación de los datos por la matriz R pasa a ser de esta forma:

Original:

```
for(i=0;i<256;i++)
  for(int j=0;j<512;j++)
    y[j] +=x_temp[i] * R[j + (i*m)];
```

Optimizado:

```
for (i=0;i<256;i++)
  for (j=0;j<12;j++)
    {
    index = R[j+i*12];
    y[index]+=(x_temp[i]*(float)0.0625);
    }
```

6.2. Eliminación del filtro harr

El algoritmo original implementa una pasada de los datos de entrada por un filtro haar destinado a wavelets de imágenes. Este tipo de filtro se aplica sobre todo para datos de 2 dimensiones o más, y en nuestro caso, una señal de electrocardiograma, cuenta con una sola dimensión, por lo que se puede eliminar del código de las funciones de A y At:

```
void A(double x[],double y[],double R[],int m,int n)
  {
  ...
  iMDWT(y_temp, 1, n, harr, 2, 9, x_temp);
  iMDWT(x_temp, 1, n, h, 10, 9, y_temp);
  ...
  }
```

6.3 Optimización de copias e inicializaciones de variables auxiliares

Todos los bucles de inicialización se sustituyen por la instrucción *memset* de c, y todas las copias de variables auxiliares se sustituyen por instrucciones *memcpy*, que son mucho más rápidas y eficientes.

Ejemplo:

```
for (i=0;i<256;i++) y[i]=0;
se sustituye por: memset(y,0,1024);

for(i=0;i<512;i++) x_temp[i] = x[i];
se sustituye por memcpy(x_temp,x,2048);
```

7. Implementación HW en detalle

Con el algoritmo ya optimizado, y haciendo profiling para ver cuales eran las funciones más utilizadas y en las que se empleaba más tiempo, se obtuvieron los siguientes resultados:

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
23.02	29.02	29.02	1424	0.02	0.03	At
20.30	54.60	25.59	51282	0.00	0.00	bpsconv
10.87	68.31	13.70	25632	0.00	0.00	fpsconv
10.65	81.73	13.42	2848	0.00	0.02	A
8.46	92.40	10.67				__divdf3
5.82	99.74	7.34				__floatsidf
4.68	105.64	5.90				__pack_d
4.28	111.04	5.40				_fpadd_parts
3.80	115.83	4.79	5698	0.00	0.01	iMDWT
2.29	118.71	2.89				__unpack_d
1.43	120.51	1.80	1	1.80	90.58	solve_L1
1.30	122.15	1.64				__muldf3
0.71	123.05	0.90	2848	0.00	0.01	MDWT
0.66	123.89	0.83	1424	0.00	0.00	Prox_L1
0.51	124.53	0.64				__adddf3
0.43	125.07	0.54	2849	0.00	0.00	norm
0.42	125.60	0.53				__fixdfsi
0.29	125.97	0.37				__subdf3
0.07	126.05	0.09				__ltdf2
0.01	126.07	0.02				__ieee754_sqrt
0.00	126.07	0.00	1	0.00	90.59	main
0.00	126.07	0.00				__fpcmp_parts_d
0.00	126.07	0.00				sqrt
0.00	126.07	0.00				__call_exitprocs
0.00	126.07	0.00				isnan
0.00	126.07	0.00				__errno
0.00	126.07	0.00				matherr
0.00	126.07	0.00				__do_global_ctors_aux

Tras estudiar el código y valorar la dificultad y viabilidad de hacer un modulo hardware para cada función, se eligieron las siguientes para su implementación: At, A, MDWT, iMDWT, bpsconv y fpsconv. Para el resto de funciones, se eligió portarlas a ensamblador, pues la ganancia de tiempo es significativa y contribuye notablemente a minimizar el tiempo de ejecución.

Además, al compilar este código en ensamblador, no genera las funciones auxiliares que se pueden ver en el profile, pues sirven para simular las carencias del microblaze por software, ya que este procesador es muy simple y no dispone de unidades avanzadas de punto flotante. Con esto, la ganancia es aún mayor.

7.1. Guía de lectura

Durante todo el apartado 8 se hacen referencias al código en su versión sotware para compararlo con la versión hardware. Para hacer una lectura más fácil y comprensible se ha seguido una regla muy simple: las referencias al código en C están en **añil** y las referencias a VHDL están en **rojo**.

7.2. fpsconv

fpsconv es junto con **bpsconv** la función que se ejecuta durante más tiempo, por tanto buscamos en ella la mayor optimización posible.

El código en C es el siguiente:

```
void fpsconv(float x_in[],int lx,const float h0[],const float
h1[],int lhm1,float x_outl[],float x_outl2[],int ic)
   {
   int i,j,ind;
   float x0,x1;
   ind=0;
   for (i=0;i<(lx);i+=2)
      {
      x0=0;
      x1=0;
      for (j=0;j<=lhm1;j++)
         {
        x0=x0+(float)x_in[(i+j)%lx]*h0[lhm1-j];
        x1=x1+(float)x_in[(i+j)%lx]*h1[lhm1-j];
         }
      x_outl[ind]= x0;//(x0/65536);
      x_outl2[ind]=x0;
      x_outl[ind+ic]=x1;//(x1/65536);
      ind++;
      }
   }
```

Para conseguir el módulo VHDL: en primer lugar definiremos los parámetros del código en C. **x_in**, **x_outl** y **x_outl2** serán memorias, las cuales están definidas en un módulo superior (**MDWT**). El acceso a las memorias desde el módulo de **fpsconv** se hace a modo de entradas (data in) y salidas (enable, adress, write enable y data out) con sus correspondientes señales.

h0 y **h1** se definen como constantes dentro del propio módulo y **lx** es simplemente una señal de entrada. **lhm1** sólo sirve para acceder a los valores de **h0** y **h1**, pero como como los tenemos almacenados en el módulo VHDL y nuestra optimización nos lo permite, podemos obviar hacer una señal. **ic** es siempre la mitad de **lx**, por lo tanto obviable también.

El funcionamiento de este módulo radica en la optimización del bucle interno. Sabiendo que el bucle se ejecuta 10 veces, se ha optado por hacer un desenrrollado de este y calcular paralelizando y de la manera más rápida posible los resultados finales de **x0** y **x1** que se guardarán en las memorias **x_outl** y **x_outl2** según proceda.

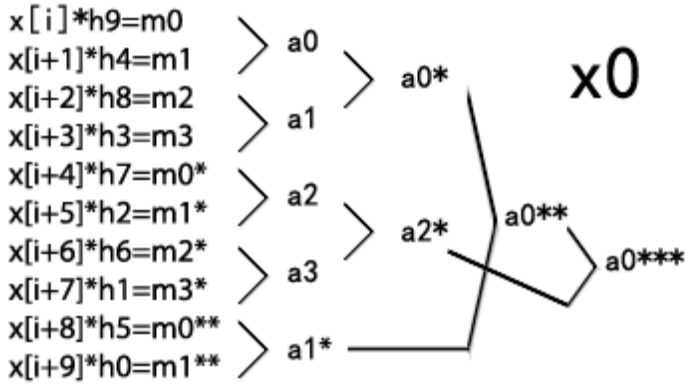
La forma de conseguir el desenrrollado en hardware de las expresiones del bucle interno (en añil aquí debajo) es el siguiente:

```
x0=x0+(float)x_in[(i+j)%lx]*h0[lhm1-j];
x1=x1+(float)x_in[(i+j)%lx]*h1[lhm1-j];
```

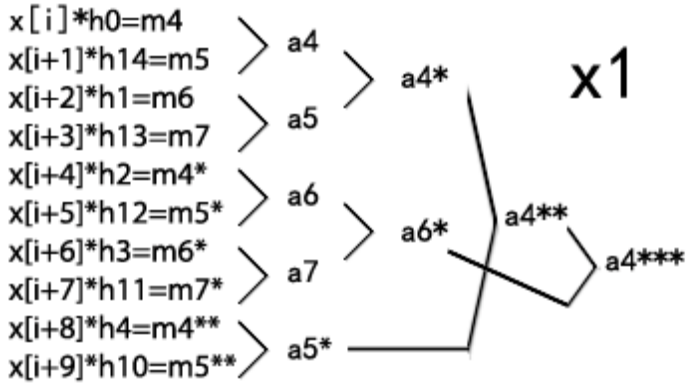
Las multiplicaciones sólo necesitan datos que ya están disponibles en las memoria **x_in** y en las constantes del filtro Daubechies (**h0** y **h1**) por lo tanto se pueden hacer 20 multiplicaciones en paralelo (2 multiplicaciones * 10 iteraciones). Pero en lugar de eso vamos a explotar los tiempos de ejecución de un multiplicador para ahorrar hardware, sin por ello renunciar a una optimización a nivel de tiempo de ejecución. Las cuatro primeras multiplicaciones para **x0** se realizan en paralelo (**x[i]*h9**, **x[i+1]*h4**, **x[i+2]*h8**, **x[i+3]*h3**) con los multiplicadores **m0**-**m3**. Análogamente se hacen las cuatro primeras multiplicaciones para **x1** con los multiplicadores **m4**-**m7** (**x[i]*h0**, **x[i+1]*h14**, **x[i+2]*h1**, **x[i+3]*h13**). Los multiplicadores se reutilizan al terminar sus cuatro primeras multiplicaciones para realizar sus cuatro restantes.

La reutilización de hardware implica que los sumadores que completan la obtención de **x0** y **x1** no sean separables de los multiplicadores. Si se separasen se perderían datos provenientes de los multiplicadores o por el contrario serían necesarios registros para almacenarlos (con la consiguiente pérdida de tiempo). Se ha optado por la primera opción: multiplicadores y sumadores funcionando en "cascada". Usaremos 8 sumadores (**a0**-**a8**), los cuales también serán reutilizables, obteniendo finalmente en la salida de los sumadores **a0** y **a4** los valores finales correspondientes a **x0** y **x1**.

En la figura se muestra un resumen del funcionamiento de los multiplicadores y los sumadores, con una reutilizacion regida por el estado en el que estemos y el ciclo por el que vayamos:



CICLO	ACCIÓN
2	inicio m0
3	inicio m1
4	inicio m2
5	inicio m3
6	inicio a0, inicio m0*
7	inicio m1*
8	inicio a1, inicio m2*
9	inicio m3*
10	inicio a2, inicio m0**
11	inicio m1***
12	inicio a0*, inicio a3
14	inicio a1*
16	inicio a2*
18	inicio a0**
22	inicio a0***
26	salida ok



CICLO	ACCIÓN
2	inicio m4
3	inicio m5
4	inicio m6
5	inicio m7
6	inicio a4, inicio m4*
7	inicio m5*
8	inicio a5, inicio m6*
9	inicio m7*
10	inicio a6, inicio m4**
11	inicio m5**
12	inicio a4*, inicio a7
14	inicio a5*
16	inicio a6*
18	inicio a4**
22	inicio a4***
26	salida ok

Además tenemos un "contador" que va actualizando los índices que hacen avanzar la función. Estos son **i** (de 2 en 2) e **ind** (de 1 en 1):

También hay una serie de lógica combinacional que calcula directamente los índices a los que se accede en la memoria **x_in**. **(i+j)** corresponde a **i+2**, **i+3**... y **(i+j)%lx** correspnde a **i2 and lx1**, **i3 and lx1**...

```
x_in[(i+j)%lx]

i2<=i+2;
// i3 a i7
i8<=i+8;
i9<=i+9;
lx1<=lx-1;

dir2<=i2 and lx1;
// i3 a i7
dir8<=i8 and lx1;
dir9<=i9 and lx1;
```

Finalmente la máquina de estados que actúa de control. En la figura sólo se indican las transiciones y su condición (si la tiene). Se incluye una descripción general de la función de cada estado. En el soporte digital se puede encontrar el código completo:

S0: estado de principio. Cambia de estado con **start**

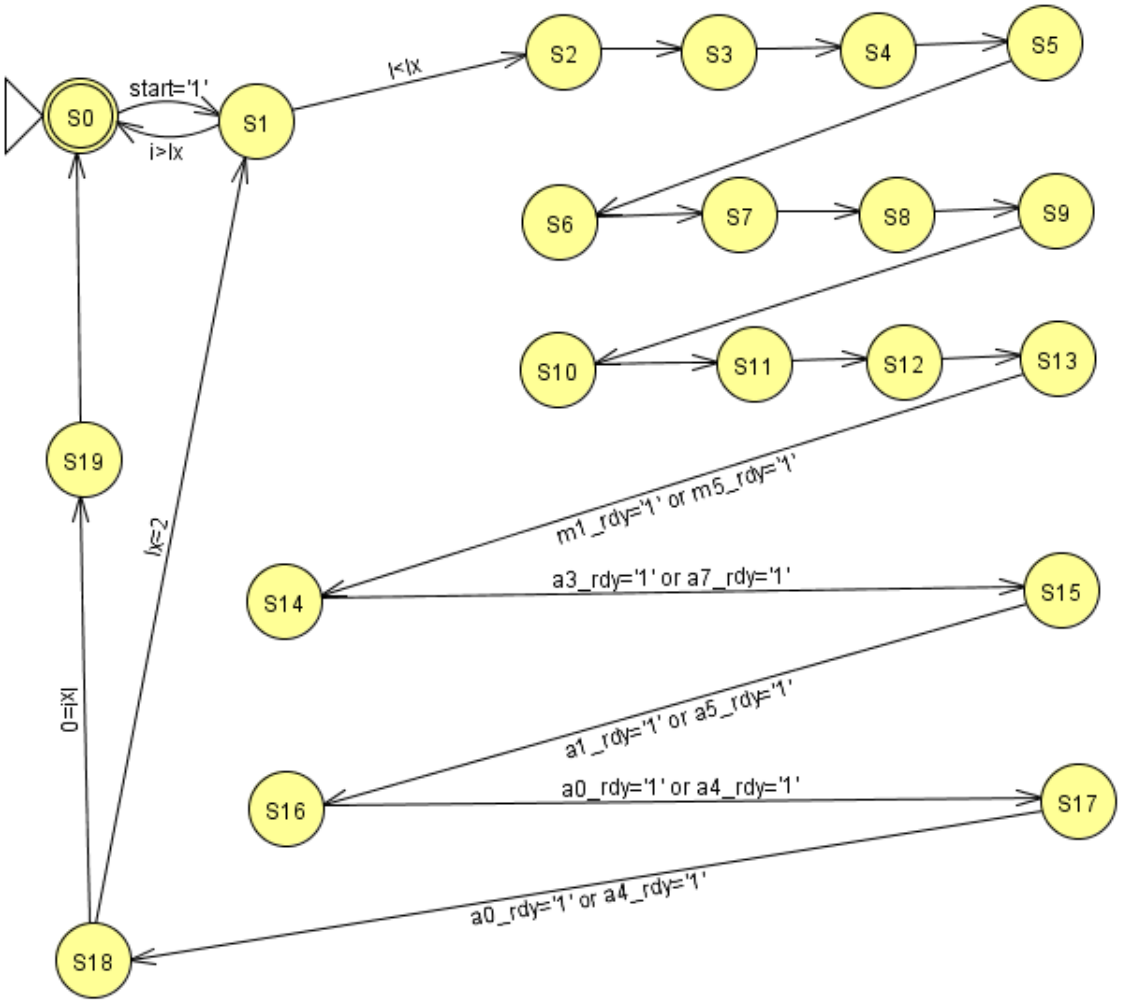
S1: control del bucle externo, decide si seguir ejecutandolo o si hemos terminado. Actúa de estado final en ese caso.

S2-**S16**: obtención de **x0** y **x1** mediante el uso de los multiplicadores y sumadores explicado anterioremente.

S17: se escribe en las memorias **x_outl2** y **x_outl** los valores **x0** y **x1** respectivamente.

S18: comprueba si estamos en el caso especial en el que **lx**=2 y si es así aún no hemos terminado y tenemos que pasar por un estado más. Si hemos terminado volvemos a **S1**.

S19: en el caso especial de **Ix**=2 se llega desde **S18** para escribir **x0** en **x_outl**.



7.3. MDWT

MDWT es junto con **iMDWT** las funciones que gestionan las llamadas a las funciones básicas que son **fpsconv** y **bpsconv** respectivamente.

El código en C es el siguiente:

```
void MDWT(float x[],int m,int n,const float h0[],const float h1[],int
lh,int L,float y[])
  {
  int actual_L,actual_n,lhm1;
  float x_temp[512];
  lhm1=lh-1;
  actual_n=512;
  for (actual_L=0;actual_L<L;actual_L++)
    {
    if (actual_L%2==0)
      fpsconv(x,actual_n,h0,h1,lhm1,y,x_temp,actual_n/2);
    else
      fpsconv(x_temp,actual_n,h0,h1,lhm1,y,x,actual_n/2);
    actual_n=actual_n>>1;
    }
  }
```

A la hora de transformar el código a VHDL: en primer lugar definiremos los parámetros del código en C. **x** e **y** serán memorias, las cuales no están definidas en el propio módulo, pero se accede a ellas a través de entradas (**Din_X**, **Din_Y**) y salidas (**En_X**, **Addr_X**, **Wen_X**, **Dout_X**, **En_Y**, **Addr_Y**, **Wen_Y** y **Dout_Y**) provinientes de un módulo superior (**At**).

h0 y **h1** no aparecen en este módulo, ya que es finalmente **fpsconv** quien las utiliza y es ese módulo el que las tiene (como hemos comentado en el apartado anterior).

m no se usa, por lo cual no será ninguna entrada. Pasa algo parecido con **n** y con **lh**. El primero es siempre 512, y el segundo es siempre 10, por tanto deducir sus valores es trivial y no precisan de entradas. **L** tampoco es una entrada ya que conocemos su valor de antemano, 9 en este caso.

El módulo contiene una memoria auxiliar llamada **x_temp** la cual guarda valores producidos por **fpsconv** en las llamadas pares para que puedan ser reutilizados en las llamadas impares.

En cuanto a optimizaciones realmente no las hay. Es una transcripción bastante fiel al código en software pero con las peculiaridades de una descripción hardware. El funcionamiento es simple, este módulo tiene una instancia de **fpsconv** con un mapeo general de señales sobre esta instancia, pero luego, dependiendo del funcionamiento derivado de las siguientes líneas el mapeo varía:

```
if (actual_L%2==0)
  fpsconv(x,actual_n,h0,h1,lhm1,y,x_temp,actual_n/2);
else
  fpsconv(x_temp,actual_n,h0,h1,lhm1,y,x,actual_n/2);
actual_n=actual_n>>1;
```

En las llamadas pares **fpsconv** lee de **X** y escribe **x0** (de **fpsconv**) en **x_temp** y **x1** (de **fpsconv**) en **Y**. En las llamadas impares **fpsconv** lee de **x_temp** y escribe **x0** (de **fpsconv**) en **X** y escribe **x1** (de **fpsconv**) en **Y**.

Esto se consigue con la lógica combinacional inferida por estas líneas de código VHDL:

```
din_xin_i <= din_temp when estado=S3 else Din_X;
En_X <= en_xout_i when estado=S3 else en_xin_i;
Addr_X <= addr_xout_i when estado=S3 else addr_xin_i;
Wen_X <= wen_xout_i when estado=S3 else wen_xin_i;
Dout_X <= dout_xout_i when estado=S3 else dout_xin_i;
en_temp <= en_xin_i when estado=S3 else en_xout_i;
addr_temp <= addr_xin_i when estado=S3 else addr_xout_i;
we_temp <= wen_xin_i(0) when estado=S3 else wen_xout_i(0);
dout_temp <= dout_xin_i when estado=S3 else dout_xout_i;
```

Además tenemos un "contador" que va actualizando los índices que hacen avanzar la función. Estos son **actual_L** (sumando 1) y **actual_n** (dividiendo por 2).

Finalmente la máquina de estados que actúa de control. En la figura sólo se indican las transiciones y su condición (si la tiene). Se incluye una descripción general de la función de cada estado. En el soporte digital se puede encontrar el código completo:

S0: estado de principio/final. Cambia de estado con **start**

S1: control del bucle, decide si seguir ejecutandolo o si hemos terminado. En caso de seguir ejecutando decide entre los estados **S2** y **S3** dependiendo de que **actual_L(4)** sea 0. En caso de haber terminado enciende la señal de **done** y vuelve a **S0**.

S2 y **S3**: esperan a que termine de ejecutar **fpsconv**.

S4: en este estado se actualizan los índices **actual_L** y **actual_n** y se vuelve directamente a **S1**.

7.4. At

At es junto con **A** las funciones que gestionan directamente la "sensing matrix".

El código en C es el siguiente:

```
void At(float x[],float y[],float R[],int m,int n)
   {
   int i,j,index;
   float x_temp[512];
   for(i=0;i<n;i++)
      {
      x_temp[i]=0;
      for(j=0;j<12;j++)
         {
         index=R[j+i*12];
         x_temp[i]+=(x[index]*(float)0.0625);
         }
      }
   MDWT(x_temp,1,n,daub0_mdwt,daub1_mdwt,10,9,y);
   }
```

A la hora de transformar el código a VHDL: en primer lugar definiremos los parámetros del código en C. **m** e **n** son los tamaños de los vectores de entrada, por tanto se pueden obviar ya que sabemos sus valores de antemano. **R** es la "sensing matrix" y junto con **x** e **y** que son las entradas se ocupan de realizar la multiplicación de la matriz. Subapartado del que a continuación detallamos el funcionamiento.

7.4.1. Multiplicación por R (mmatrixf)

x, **y** y **R** pasan a ser una serie de señales que representan memorias a través de entradas (**Din_X**, **Din_Y**, **Din_R**) y salidas (**En_X**, **Addr_X**, **Wen_X**, **En_Y**, **Addr_Y**, **Wen_Y** y **Dout_Y**) provinientes de **At**.

La particularidad a la hora de hacer esto en VHDL radica en estas líneas:

```
for(j=0;j<12;j++)
   {
   index=R[j+i*12];
   x_temp[i]+=(x[index]*(float)0.0625);
   }
```

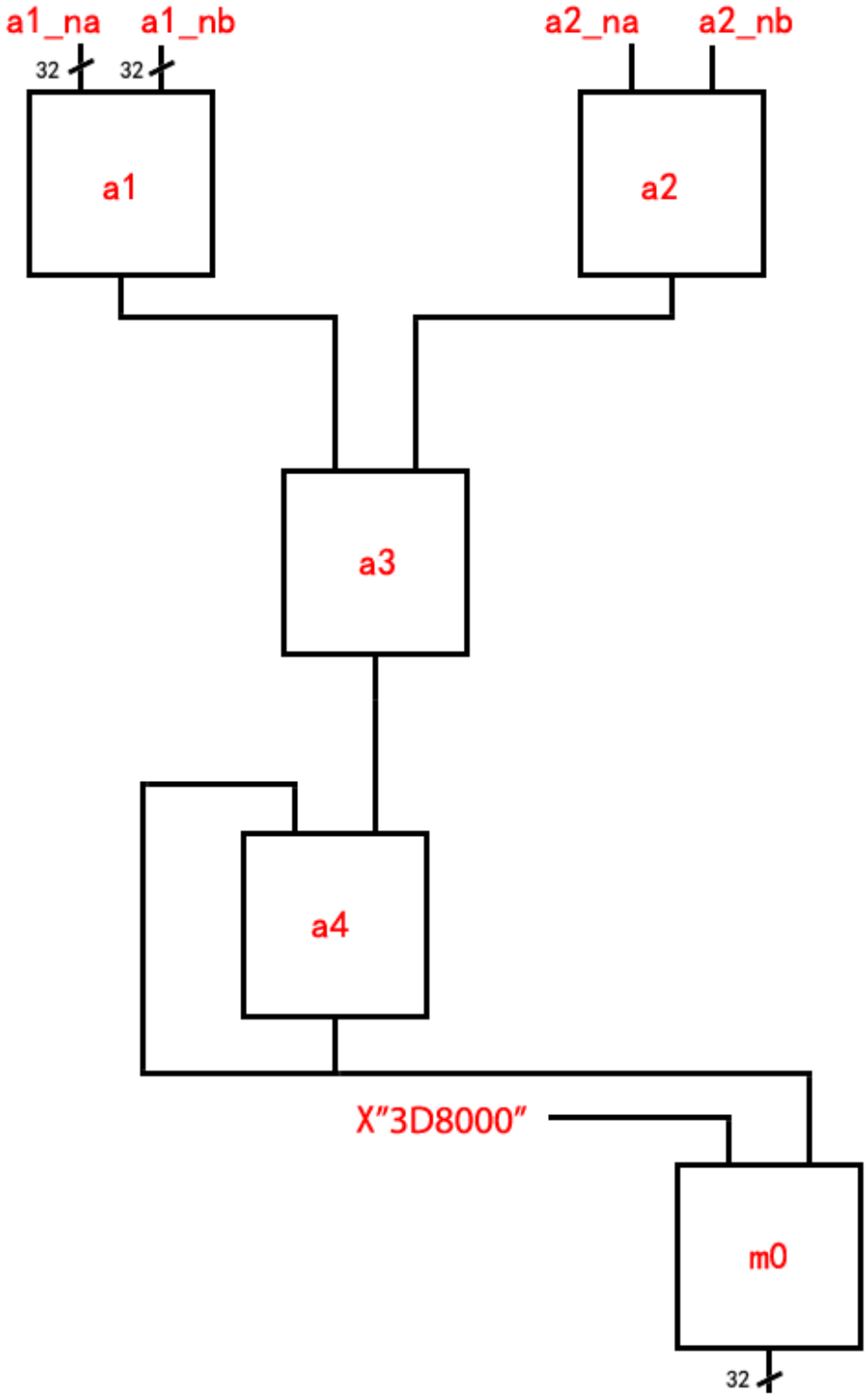
El bucle en **j** del software se desenrrolla tres veces porque en cada lectura de **R** se obtienen cuatro índices (4 índices * 3 veces desenrrollado = 12 índices) ya que llegan 32 bits y cada índice ocupa un byte en **R**.

Con los índices obtenidos accedemos a cuatro datos de **x** tal que así: **x[primer byte de R]**, **x[segundo byte de R]**... y así hasta el cuarto, metiéndolos como entradas de los sumadores correspondientes **a1**-**a2**:

```
when S3=>
   En_X<='1';
   Addr_X<=BASE_ADDR_X+('0'&Din_R(0 to 7)&"00");  // Primer byte
   nestado<=S4;
when S4=>
   a1_na<=Din_X;  // Obtención del dato
   En_X<='1';
```

Los resultados de los sumadores **a1**-**a2** se sumarán en **a3** y finalmente se acumula en **a4** para la siguiente iteración. Además el resultado de **a4** se

manda al multiplicador **m0** para conseguir la multiplicación por 1/16. A continuación la representación de los sumadores:



Además tenemos un "contador" que va actualizando los índices que hacen avanzar la función. Estos son **j** (de 4 en 4 por el desenrrollado del bucle) e **i** (de 1 en 1).

Finalmente la máquina de estados que actúa de control. En la figura sólo se indican las transiciones y su condición (si la tiene). Se incluye una

descripción general de la función de cada estado. En el soporte digital se puede encontrar el código completo:

S0: estado de principio/final. Cambia de estado con **start**

S1: control del bucle externo, decide si seguir ejecutandolo bajo la condición de **i<512**. En caso de hacerlo pasa **S2**, en caso contrario vuelve a **S0** indicando que hemos termiando. También se ocupa de que la primera entrada de **a4** sea 0.

S2: control del bucle interno, decide si seguir ejecutandolo bajo la condición de **i<12**. En caso de hacerlo obtiene cuatro índices de la matriz **R**. En caso de no hacerlo, salta directamente a **S12**.

S3-**S11**: suma los datos obtenidos y los acumula en **a4** para la siguiente iteración.

S12: prepara el multiplicador para que se ejecute y pasa a **S13**.

S13: espera al resultado del multiplicador y lo guarda en **y**. Salta a **S14**.

S14: vuelve a **S1**.

7.4. At (continuación)

Una vez detallado el módulo **mmatrixf** podemos continuar explicando **At**, que incluirá obviamente, una instancia de **mmatrixf**.

También se instancia una memoria llamada **temp1** que corresponde con la **x_temp** del código y por tanto es donde se guardan los datos salientes del módulo **mmatrixf**.

Por supuesto **MDWT** también está presente con su correspondiente instancia, ya que usará **x_temp** para realizar sus operaciones.

La máquina de estados es muy simple, se limita a activar el módulo **mmatrixf** que escribe en **x_temp** y cuando acaba esos datos son usados por **MDWT**:

S0: estado de principio/final. Cambia de estado con **start** y activa el módulo **mmatrixf** con **start_mat=1**.

S1: espera a que **mmatrixf** termine, y cuando se confirma activa el módulo **mdwt_d** saltando a **S2**.

S2: espera a que termine **mdwt_d** y una vez está confirmado vuelve a **S0** para finalizar.

7.5. bpsconv

bpsconv es junto con **fpsconv** la función que se ejecuta durante más tiempo, por tanto buscamos en ella la mayor optimización posible.

El código en C es el siguiente:

```
void bpsconv(float x_out[], int lx, const float g0[],const float
g1[], float x_inl[], float x_inh[])
  {
  int i,j,ind,tj,tjl,indice;
  float x0,x1;
  ind=0;
  for (i=0;i<lx;i++)
    {
    x0=0;
    x1=0;
    tj=0;
    tj1=1;
    for (j=4;j>=0;j--)
      {
      if (lx>2)
        indice=(i+j-4+lx)%lx;
      else
        indice=(i+j)%lx;
      x0=x0+x_inl[indice]*g0[tj]+x_inh[indice+lx]*g1[tj];
      x1=x1+x_inl[indice]*g0[tjl]+x_inh[indice+lx]*g1[tjl];
      tj+=2;
      tjl+=2;
      }
    x_out[ind]=x0;
    x_out[ind+1]=x1;
    ind+=2;
    }
  }
```

Para conseguir el módulo VHDL: en primer lugar definiremos los parámetros del código en C. **x_out**, **x_inl** y **x_inh** serán memorias (usando como entradas y salidas –según corresponda- señales de enable, adress, data in, write enable y data out), que a su vez son accedidas por el módulo superior (**iMDWT**).

g0 y **g1** se definen como constantes dentro del propio módulo y **lx** es simplemente una señal de entrada.

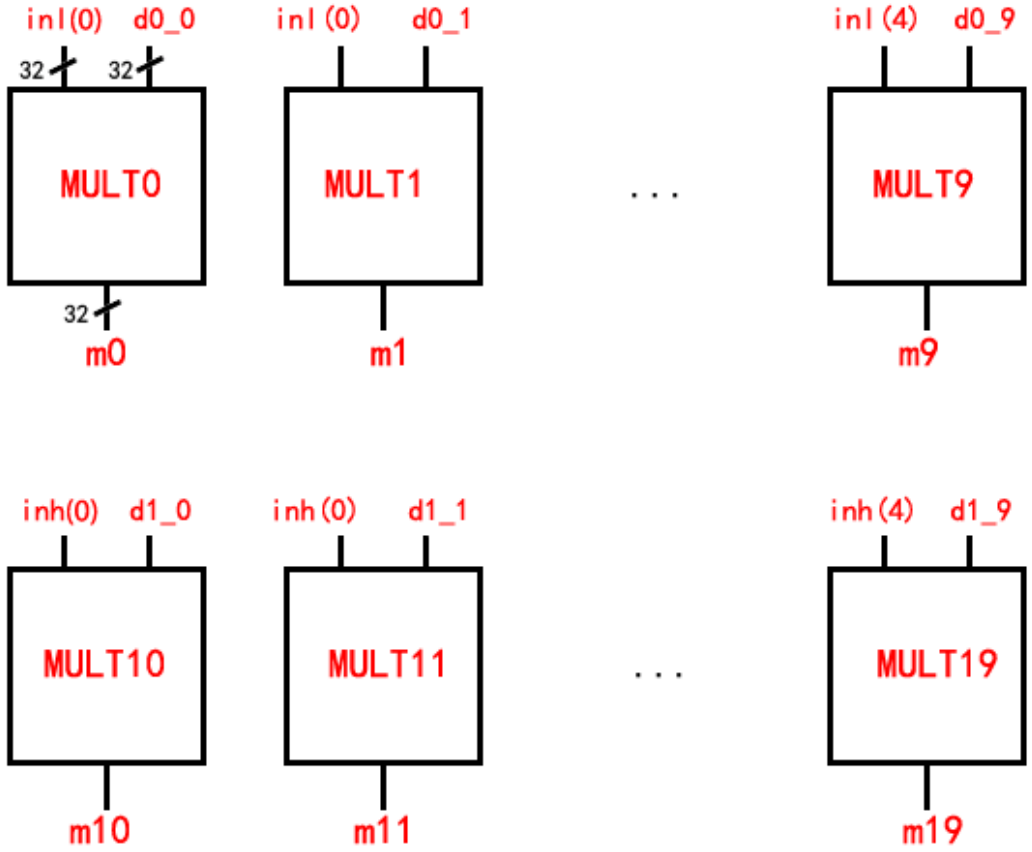
La optimización de este módulo radica primero en la optimización del código en software, que ha llevado a mejorar el bucle interno reduciendo variables y mejorando una descripción bastante ofuscada. Una vez conseguido esto y sabiendo que el bucle se ejecuta 5 veces, se ha optado por hacer un desenrrollado de este y calcular paralelizando y de la manera más rápida posible los resultados finales de **x0** y **x1** que se guardarán en la memoria correspondiente a **x_out**.

La forma de conseguir el desenrrollado en hardware de las expresiones del bucle interno (en añil aquí debajo) es el siguiente:

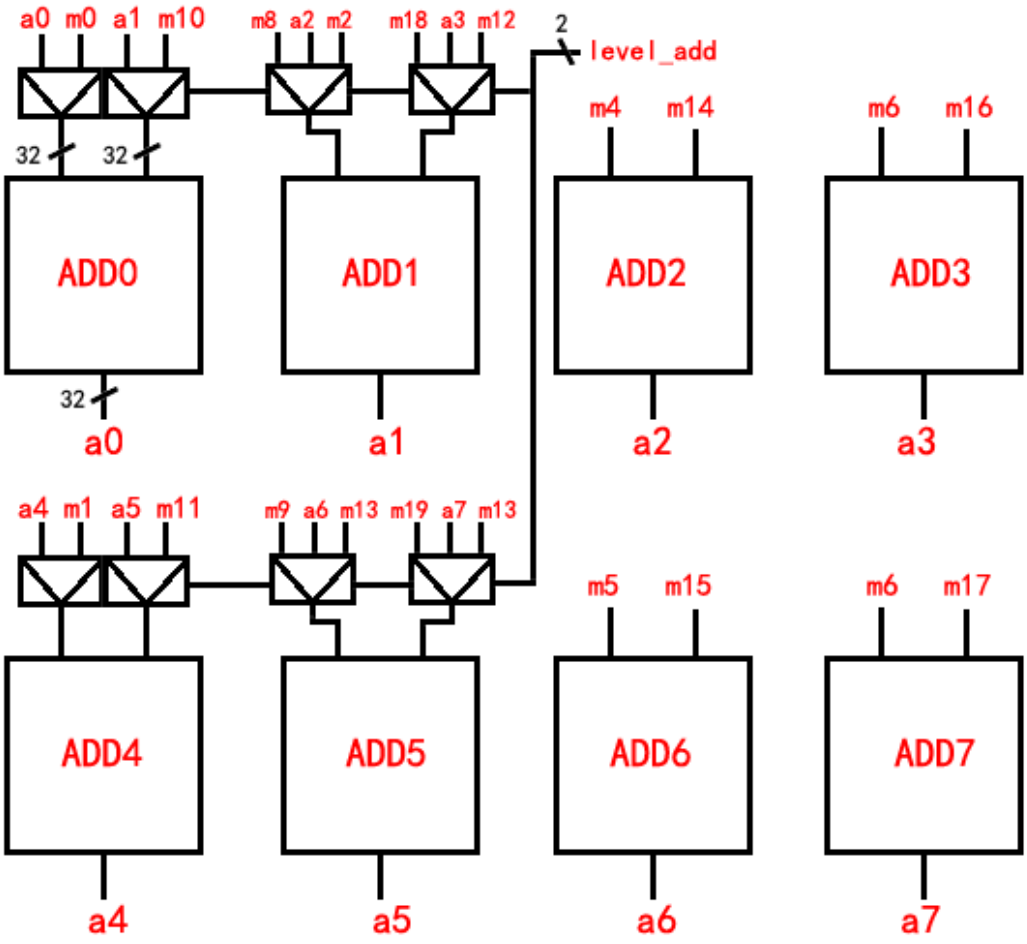
```
x0=x0+x_inl[indice]*g0[tj]+x_inh[indice+lx]*g1[tj];
x1=x1+x_inl[indice]*g0[tjl]+x_inh[indice+lx]*g1[tjl];
```

Las multiplicaciones sólo necesitan datos que ya están disponibles en las memorias (**x_inl** y **x_inh**) y en las constantes del filtro Daubechies (**g0** y **g1**) por lo tanto se usan 20 multiplicadores en paralelo (4 multiplicaciones * 5

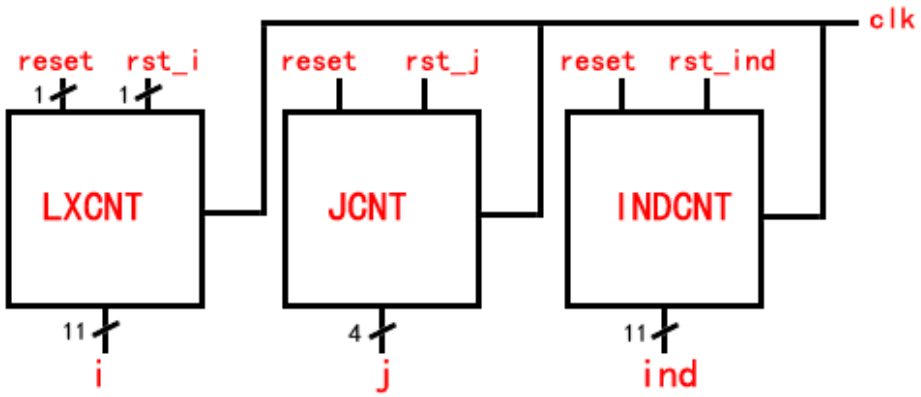
iteraciones) que se ocupan de sacar los productos que será necesario sumar después. En la figura, se representan los multiplicadores:



Para realizar las sumas del bucle desenrrollado usaremos 8 sumadores de los cuales 4 serán reutilizables funcionando según lo que nos indique la señal **level_add**. En la figura, los sumadores con sus correspondientes multiplexores:



Para cambiar los índices de los bucles (**i**, **j**) y el cambio de **ind** se tienen tres contadores que suman (o restan) según lo indique la señal correspondiente (**i_en**, **j_en**, **ind_en**) controlada por la máquina de estados:



También hay una serie de lógica combinacional (además de los multiplexores a la entrada de los sumadores) que calcula **indice** (**index**) directamente, para evitar tener más estados en la máquina de estados:

```
if (lx>2)
  indice=(i+j-4+lx)%lx;
else
  indice=(i+j)%lx;

index<= (i+j+lx_reg-"0100") and (lx_reg-1) when lx_reg>2
  else (i+j) and (lx_reg-1);
```

Finalmente la máquina de estados que actúa de control. En la figura sólo se indican las transiciones y su condición (si la tiene). Se incluye una descripción general de la función de cada estado. En el soporte digital se puede encontrar el código completo:

idle: estado de principio-final. Cambia de estado con **start**

s0: control del bucle externo, decide si seguir ejecutandolo o si hemos terminado.

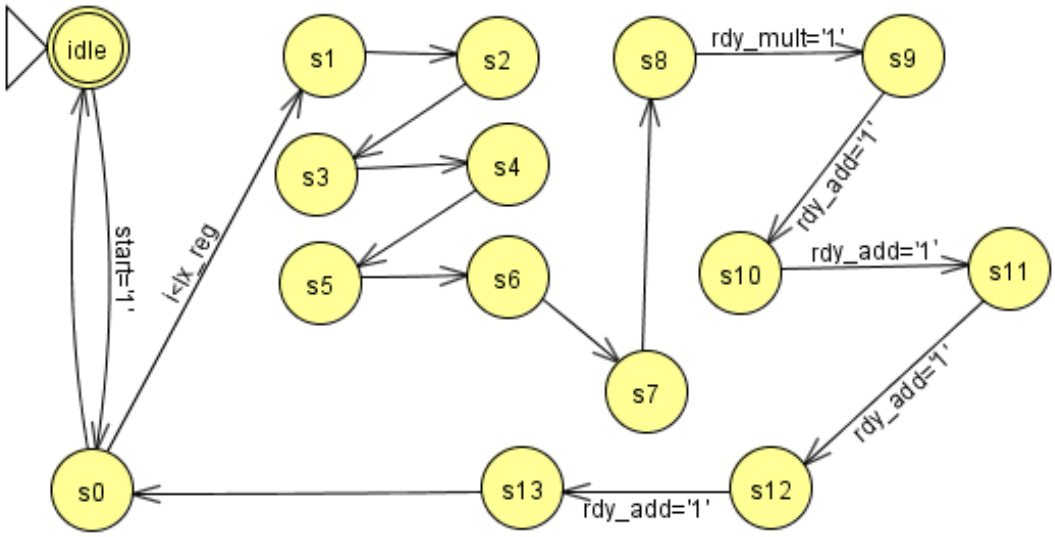
s1-**s6**: preparan datos para los multiplicadores.

s7: pone en funcionamiento los multiplicadores.

s8: comprueba la finalización de los multiplicadores y en ese caso pone en funcionamiento los sumadores.

s9-**s11**: uso y reutilización de los sumadores.

s12-**s13**: almacenamiento de los datos en la memoria y en **s13** vuelta al **s0** para comprobar si se sigue o no en el bucle.



7.6. iMDWT

iMDWT es junto con **MDWT** las funciones que gestionan las llamadas a las funciones básicas que son **bpsconv** y **fpsconv** respectivamente.

El código en C es el siguiente:

```
void iMDWT(int x[],int m,int n,const int g0[],const int g1[],int lh,
int L,int y[])
  {
  int ximpar[256];
  int actual_n=1;
  ximpar[0]=y[0];
  for (int act_L=0;act_L<L;act_L++)
    {
    if (act_L%2==0)
      bpsconv_daub(x,actual_n,g0,g1,ximpar,y);
    else
      bpsconv_daub(ximpar,actual_n,g0,g1,x,y);
    actual_n = actual_n<<1;
    }
    // actual_n se multiplica por 2 8 veces-> en la ultima
    // iteración es 2^8=256
  }
```

A la hora de transformar el código a VHDL: en primer lugar definiremos los parámetros del código en C. **x** e **y** serán memorias, las cuales no están definidas en el propio módulo, pero se accede a ellas a través de entradas (**INPUT_DIN**, **OUTPUT_DIN**) y salidas (**INPUT_EN**, **INPUT_ADDR**, **INPUT_WEN**, **OUTPUT_DOUT**, **OUTPUT_EN**, **OUTPUT_ADDR**, **OUTPUT_WEN**) provinientes de un módulo superior (**A**).

g0 y **g1** no aparecen en este módulo, ya que es finalmente **bpsconv** quien las utiliza y es ese módulo el que las tiene (como hemos comentado en el apartado anterior).

m no se usa, por lo cual no será ninguna entrada. Pasa algo parecido con **n** y con **lh**. El primero es siempre 512, y el segundo es siempre 10, por tanto deducir sus valores es trivial y no precisan de entradas. **L** tampoco es una entrada ya que conocemos su valor de antemano, 9 en este caso.

En cuanto a optimizaciones realmente no las hay. Es una transcripción bastante fiel al código en software pero con las peculiaridades de una descripción hardware. El funcionamiento es simple, este módulo tiene una instancia de **bpsconv** con un mapeo general de señales sobre esta instancia, pero luego, dependiendo del funcionamiento derivado de las siguientes líneas el mapeo varía:

```
if (act_L%2==0)
  bpsconv_daub(x,actual_n,g0,g1,ximpar,y);
else
  bpsconv_daub(ximpar,actual_n,g0,g1,x,y);
actual_n = actual_n<<1;
```

En las llamadas pares **bpsconv** toma de entrada la memoria auxiliar y escribe **x0** (de **bpsconv**) en la memoria externa y **x1** (de **bpsconv**) en **Y**. En las llamadas impares **bpsconv** toma de entrada la memoria externa y escribe **x0** (de **bpsconv**) en la memoria auxiliar y escribe **x1** (de **bpsconv**) en **Y**.

Esto se consigue con la lógica combinacional inferida por estas líneas de código VHDL:

```
-- mux de addr para bpsconv_daub y XIMPAR2
ximpar2_en<= inl_en when l(4)='0' else xout_en
ximpar2_addr<= inl_addr when l(4)='0' else xout_addr;
ximpar2_wen<= inl_wen when l(4)='0' else xout_wen;

inl_din<= ximpar2_din when l(4)='0' else output_din;

-- mux de addr para bpsconv_daub y OUTPUT
output_en<= inl_en when l(4)='1' else xout_en;
  output_addr<= inl_addr+BASE_ADDR_OUTPUT when l(4)='1' else
xout_addr+BASE_ADDR_OUTPUT;
output_dout<= xout_dout;
output_wen<= inl_wen when l(4)='1' else xout_wen;
```

Además tenemos dos contadores (**LCNT** y **ACT_N**) que van actualizando los índices que hacen avanzar la función. Estos son **actN** y **L**.

Finalmente la máquina de estados que actúa de control. En la figura sólo se indican las transiciones y su condición (si la tiene). Se incluye una descripción general de la función de cada estado. En el soporte digital se puede encontrar el código completo:

idle: estado de principio/final. Cambia de estado con **start**

s0: estado en el que simplemente se copia **input[0]** en **ximpar[0]**. Pasa al estado **s1**.

s1: pone **bpsconv** en funcionamiento y salta a **s2**.

S4: se comprueba si se ha terminado (**l="01000"**) y si es así vuelve a idle y pone done a 1. En otro caso indica a los contadores que aumenten **l** y **actN**

7.7. A y 7.7.1. Multiplicación por R

La gestión que **A** hace de la matriz R es prácticamente igual a la de **At**, y la implementación en hardware difiere en detalles muy pequeños, con lo cual no merece la pena ahondar más en este módulo, ya que sería redundante.

7.8. IPCores utilizados

Floating-Point Operator v5.0 DS335

El módulo de punto flotante de Xilinx permite realizar operaciones aritméticas con valores flotantes en una FPGA. La funcionalidad del módulo (operación a realizar) se especifica cuando el módulo es generado, de forma que habrá que generar tantos módulos como tipos de operaciones se deseen realizar. Cada variante tiene una interfaz común.

Operaciones soportadas:

- ♦ Multiplicación
- ♦ Suma/resta
- ♦ División
- ♦ Raíz cuadrada
- ♦ Comparación
- ♦ Conversión de floating-point to fixed-point
- ♦ Conversión de fixed-point to floating-point
- ♦ Conversión entre tipos de floating-point

Método de redondeo:

Sólo está soportado el modo "Redondeo al más cercano" (definido por el estándar IEEE-754).

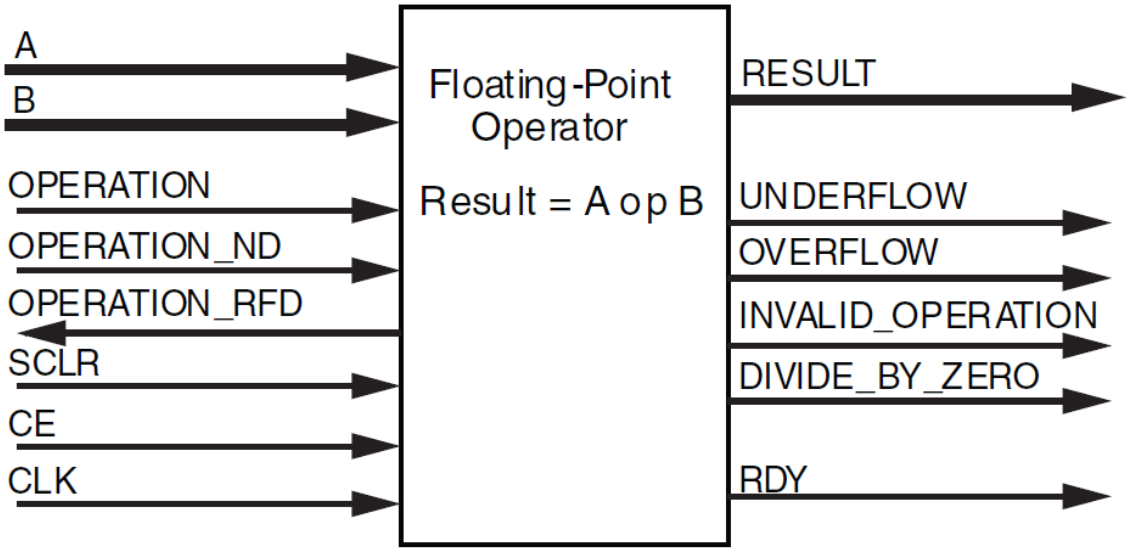


Ilustración 5 - Interfaz de la FPU

7.9. Integración del periférico en el sistema

7.9.1. Creación de un registro para el bus PLB

Para que la ejecución del periférico sea activada por el Microblaze, debe existir un canal de comunicación entre ambos. Ésta canal de comunicación es el bus PLB. Para ello se ha desarrollado un módulo VHDL que encapsula el periférico y permite conectarlo al bus como esclavo, por lo que atenderá las peticiones del procesador.

En nuestro diseño, este módulo de encapsulamiento se ha pensado como dos registros: uno de activación y otro de estado. De modo que para activar la ejecución del módulo, el procesador debe escribir cualquier valor en el registro (que desde el Microblaze se trata como una dirección de memoria de 32 bits), y para comprobar si ha finalizado, basta con realizar una espera activa leyendo el valor del registro de estado hasta que se obtiene el valor parada.

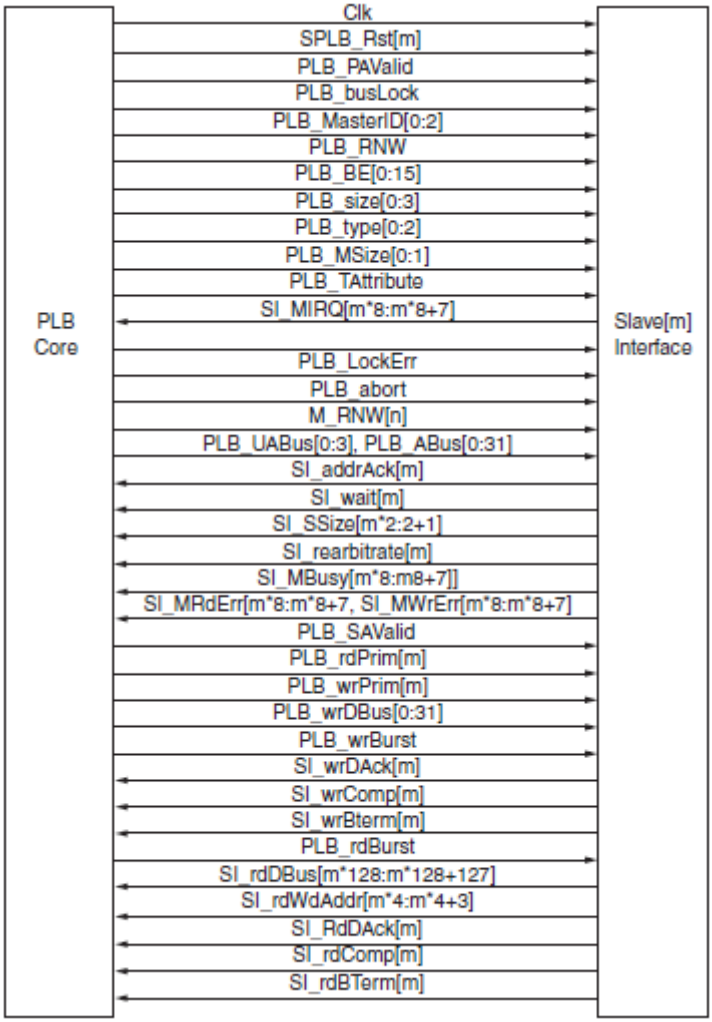


Ilustración 6 - Interfaz esclavo PLB

Como se puede ver en la imagen anterior, un módulo esclavo del PLB tiene una gran cantidad de señales, pero para un registro simple, como es nuestro caso, se necesita controlar mediante una máquina de estados simplemente estas señales:

Señales del maestro del PLB:

- **PLB_PABus**: Indica la dirección a la que se dirige la petición del maestro.
- **PLB_PAValid**: Indica que se ha realizado una petición.
- **PLB_RNW**: Indica el tipo de petición (1 lectura, 0 escritura).
- **PLB_wrDBus**: Contiene los datos que se envían desde el maestro al esclavo.

Señales del esclavo del PLB:

- **Sl_addrAck** : Indica que el esclavo al que va dirigida la petición va a atender.
- **Sl_rdDBus**: El esclavo deposita en esta señal los datos que el maestro quiere leer.
- **Sl_rdDAck**: El esclavo indica que la petición de lectura se ha completado correctamente.
- **Sl_rdComp**: El esclavo indica que está atendiendo la petición de lectura.
- **Sl_wrDAck**: El esclavo indica que la petición de escritura se ha completado correctamente.
- **Sl_wrComp**: El esclavo indica que está atendiendo la petición de escritura.
- **Sl_wait** : Indica que el esclavo al que va dirigida la petición pide el acceso al PLB.

Los siguientes cronogramas, presentan el comportamiento de dichas señales ante una petición de lectura y otra de escritura a un periférico esclavo del PLB con dirección 40000A4:

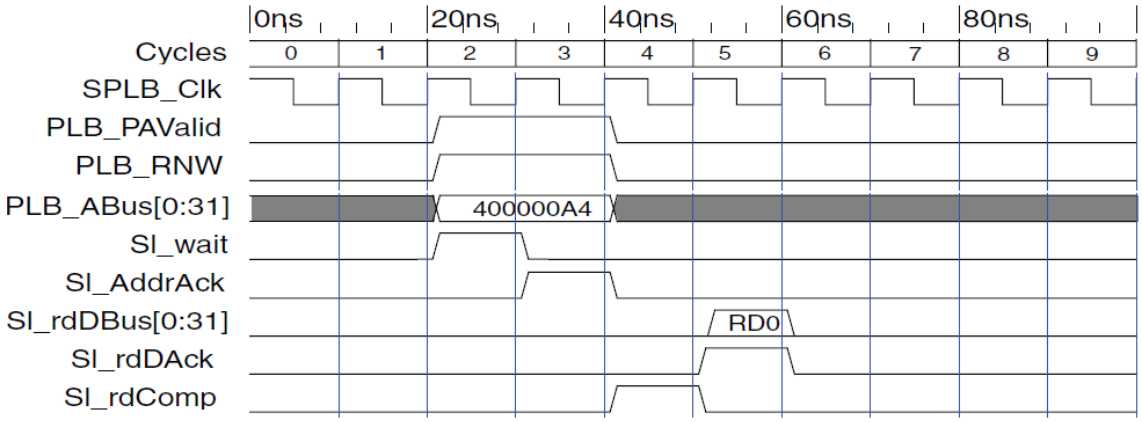


Ilustración 7 - Petición de lectura

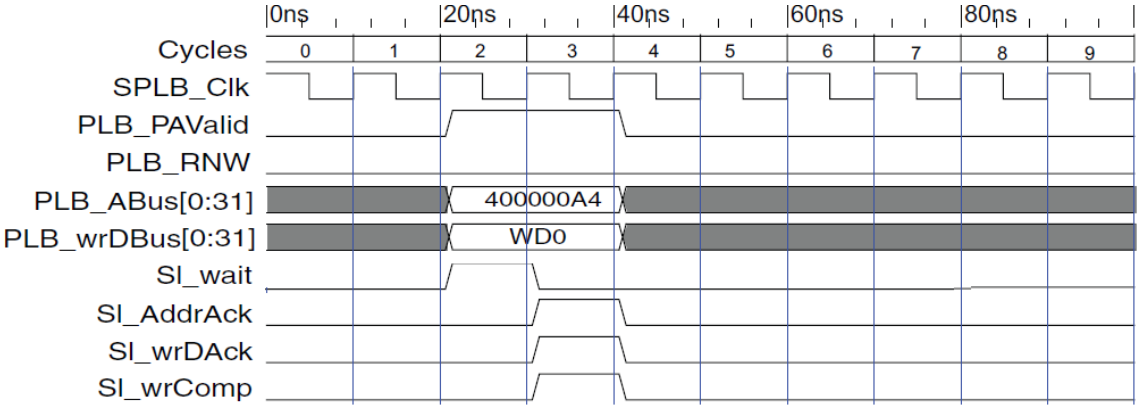


Ilustración 8 - Petición de escritura

Cabe destacar, que a la vista de los cronogramas anteriores, una petición de lectura tarda 1 ciclo extra con respecto a una petición de escritura, debido a que hay que enviar el dato pedido al bus y activar Sl_rdDAck durante el ciclo siguiente a la activación de Sl_rdComp, mientras que para una escritura, ambas señales y el dato se pueden activar/obtener en el mismo ciclo.

Una vez encapsulado, el módulo resultante tiene comunicación con el Microblaze a través de la interfaz PLB y también con las memorias de las que obtiene los datos y escribe los resultados.

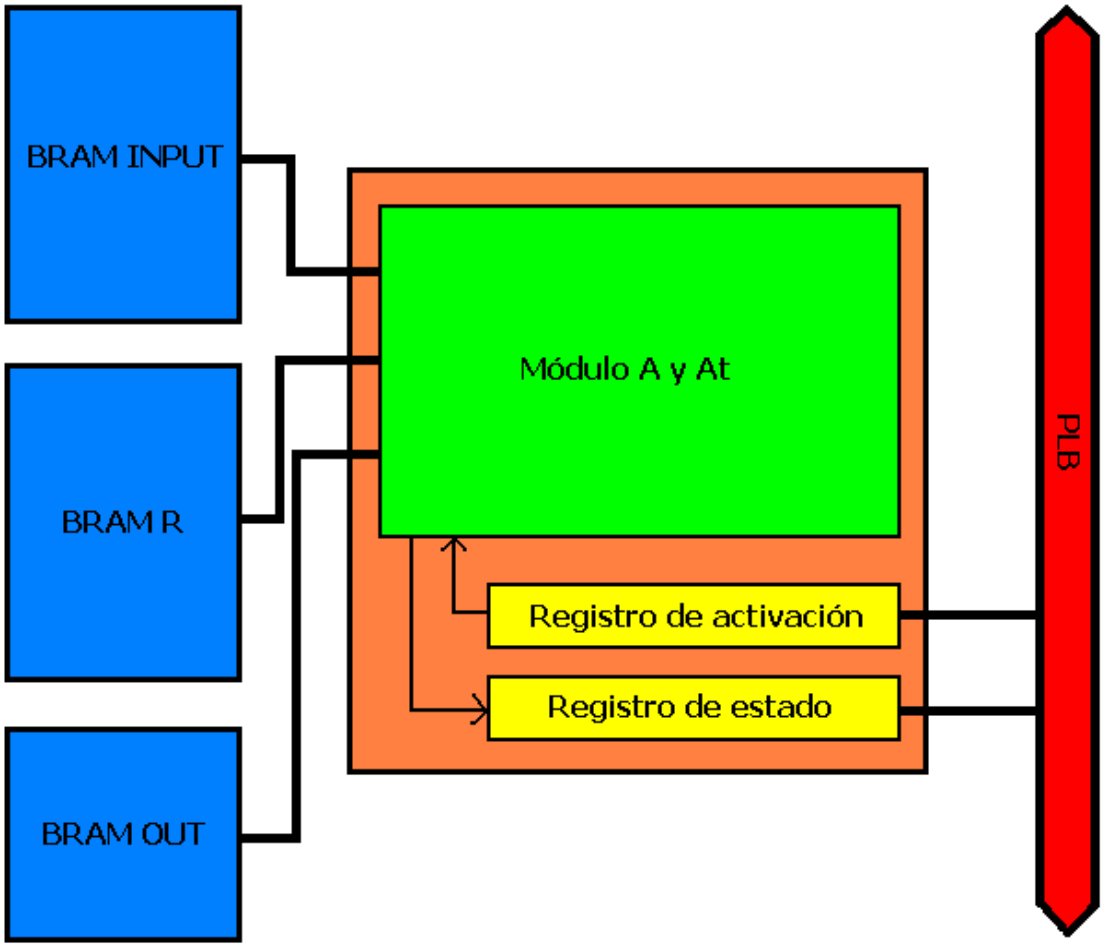


Ilustración 9 - Módulo interfaz PLB

7.9.2. Importación del periférico en el XPS

Para importar un periférico propio y poder utilizarlo en el XPS, lanzamos el asistente de importación mediante la opción del menú: Hardware -> Create or Import Peripheral. Este asistente nos va guiando por una serie de fases:

Fase 1: Origen de los datos

En esta fase indicamos que queremos importar un periférico existente a los repositorios del proyecto actual. Posteriormente nos pedirá el nombre del módulo de más alto nivel del periférico (top level module) así como todos los archivos .vhd, incluyendo incluso los de los IP cores utilizados (en este caso, las unidades de punto flotante). Es importante que seleccionemos también la importación de archivos de netlist para las unidades de punto flotante, ya que al ser IP cores externos, el XPS no es capaz por sí solo de generar dichos archivos.

Fase 2: Seleccionar la interfaz de conexión

Indicamos que se trata de una interfaz esclava para el PLB (SPLB), y el asistente se encarga de extraer todas las señales necesarias para esta conexión, generando un error en caso de no encontrar todos los puertos necesarios. A continuación se configuran los puertos que generan excepciones (ninguno en nuestro caso, ya que utilizaremos espera activa) y también se configuran los puertos genéricos que así lo requieran.

Si todo ha ido bien, aparecerá en el apartado de Local Pcores, dentro del IPCatalog de nuestro proyecto, el periférico indicado listo para ser añadido.

7.10. Código y gestión del MicroBlaze

Ya que las funciones que empleaban mas tiempo se han implementado en módulos hardware, al Microblaze le queda el trabajo de gestionar la activación de estas funciones, hacer las operaciones de actualizaciones de variables, y aquellas funciones que no se han implementado en hardware.

Se explica a continuación el funcionamiento de la gestión del periférico, el acceso a los datos que este arroja, así como las optimizaciones del código en ensamblador, y las partes que no se cambiaron, pues no merecía la pena.

7.10.1. Gestión de memoria y periféricos

El sistema utiliza memorias BRAM de 2 puertos de lectura y escritura, uno de ellos es ocupado por el modulo hardware que contiene todas las funciones, el otro puerto es ocupado por un bus de memoria local al que se conecta el Microblaze. Como se detallo en el punto anterior, el sistema emplea 3 memorias que están compartidas como se acaba de explicar.

Cuando los módulos hardware terminan su trabajo, el Microblaze puede trabajar sobre estos datos, y la forma de acceder a ellos es mediante punteros, inicializados directamente a la dirección base de cada una de las zonas de memoria sobre las que necesiten trabajar. Para minimizar el uso de memorias, en cada una hay varias variables, cada una tiene un desplazamiento desde la dirección base.

```
#define BASE_ADDR_W 0x00100000
#define BASE_ADDR_XHAT 0x00100800
#define BASE_ADDR_Y 0x00101000
#define BASE_ADDR_XOLD 0x00101400
#define BASE_ADDR_TEMP 0x00200000
#define REG_ACT 0x00300000
#define REG_STATUS 0x00300004
```

Desplazamientos de cada una de las variables utilizadas.

```
float* w=(float*)BASE_ADDR_W;
float* temp=(float*)BASE_ADDR_TEMP;
float* xold=(float*)BASE_ADDR_XOLD;
int* reg_act=(int*)REG_ACT;
int* reg_status=(int*)REG_STATUS;
```

Inicializaciones de los punteros que hacen los accesos.

El periférico tiene además dos modos de funcionamiento, en uno de ellos ejecuta las funciones A y At (con todas las funciones que ello conlleva, mirar punto 6.2), y en el otro modo se ejecuta solamente A, pero tomando los datos de entrada de otra zona de memoria. Esto se gestiona desde el software mediante la escritura de un 0 o un 1 en su registro de activación (desplazamiento 0 desde la dirección base asignada al periférico).

Para saber si el periférico ha terminado su ejecución, basta con leer un 0 en su registro de estado, pues si se lee un 1 significa que aun esta en ejecución (el registro de estado tiene desplazamiento +4 desde la dirección base del periférico). Para la comprobación de la finalización de ejecución del hardware se realizan esperas activas, pues hasta que no hay nuevos datos

disponibles no se puede trabajar sobre ellos, y el procesador no se puede ocupar en otras tareas, pues no hay nada que hacer.

```
*reg_act=1;
while (*reg_status==1){}
```

Activación del periférico en modo 1 y espera activa para su finalización.

7.10.2. Código en ensamblador

Con las funciones anteriormente mencionadas ya en hardware, aun quedan varias funciones que hay que hacer: norm, proxL1, y los bucles que se encuentran dentro de solve_L1. En este punto se procedió a una nueva optimización del código, tratando de eliminar el mayor numero de bucles consecutivos posibles. Con esta intención, se consiguió utilizar un solo bucle para realizar la actualización de xhat (en este punto, salida de la primera ejecución del hardware en modo 0), la función proxL1, la norma de chat, y la actualización de w.

```
for (i=0;i<n;i++)
      {
      xhat[i]=xhat[i]*(2.0/L)+w[i];
      signxhat=sign(xhat[i]);
      xhat[i]=fabs(xhat[i])-(lambda/L);
      if (xhat[i]<=0) xhat[i]=0;
      else xhat[i]=xhat[i]*signxhat;
      w[i]=xhat[i]+((told-1)t)*(xhat[i]-xold[i]);
      sum+=fabs(xhat[i]);
      }
```

Optimización de bucles.

El bucle de la norma de temp (en este punto, salida de la segunda ejecución del periférico en modo 1), no se pudo optimizar de ninguna manera, y quedó como estaba.

Una vez hecha esta optimización, se procedió a portar el código a ensamblador. La aparición del modificador volatile tiene su razón en la utilización de optimizaciones del compilador, añadiendo volatile se suprime la posibilidad de que el compilador coloque la instrucción en otro orden, arruinando así la ejecución. De esta forma se elimina también la inclusión de la librería math.h, además de la omisión de la definición de macros en el fichero de cabeceras. Según se puede comprobar en la documentación del Microblaze, los registros que están disponibles para propósito general son r19-r31, por lo que se ha restringido el uso a estos.

Para el cálculo de t se necesitan multiplicaciones, divisiones y raíces en punto flotante. Para esto se utiliza la unidad de punto flotante extendida del Microblaze, que da soporte para estas operaciones, pero no están segmentadas, de modo que el procesador las ejecuta en orden y debe esperar a que acabe una instrucción para comenzar la siguiente.

```
t = (1+sqrt(1+4*told*told))/2;
```

Código original para el cálculo de t.

```
asm volatile("fmul r20,%0,%1"::"r"(told),"r"(told));
asm volatile("fmul r21,r20,%0"::"r"(4.0F));
asm volatile("fadd r20,r21,%0"::"r"(1.0F));
asm volatile("fsqrt r21,r20");
asm volatile("fadd r22,r21,%0"::"r"(1.0F));
asm volatile("fdiv %0,%1,r22":"=r"(t):"r"(2.0F));
```

Código en ensamblador para el cálculo de t.

Como se puede ver en el código, se utilizan restricciones (los operadores %) para tener una relación directa entre las variables definidas en C y los registros internos del Microblaze, sin necesidad de almacenar los registros en memoria y cargarlos con un puntero.

Para el bucle antes descrito se emplea un número mayor de registros, además de hacer uso de las instrucciones de salto, e instrucciones de aritmética entera, así como aritmética en punto flotante. Para aumentar el número de instrucciones ejecutadas por unidad de tiempo, se utiliza el delay slot que ofrece la arquitectura del Microblaze. De esta forma, en los saltos, siempre se ejecuta la instrucción inmediatamente a continuación, que se ha aprovechado para actualizar el índice i del bucle.

```
sum=0;
for (i=0;i<n;i++)
      {
      xhat[i]=xhat[i]*(2.0/L)+w[i];
      signxhat=sign(xhat[i]);
      xhat[i]=fabs(xhat[i])-(lambda/L);
      if (xhat[i]<=0) xhat[i]=0;
      else xhat[i]=xhat[i]*signxhat;
      w[i]=xhat[i]+((told-1)t)*(xhat[i]-xold[i]);
      sum+=fabs(xhat[i]);
      }
```

Código original en C.

Como ejemplo, no se muestra aquí todo el código correspondiente al bucle en ensamblador, pero si una muestra significativa que da una idea general de cómo se ha hecho el resto.

```
asm volatile("add r20,r0,%0"::"r"(n*4-4)); // i
asm volatile("fdiv r21,%0,%1"::"r"(L),"r"(2.0F)); // 2.0/L
asm volatile("fdiv r22,%0,%1"::"r"(L),"r"(lambda)); // lambda/L
asm volatile("fadd r26,%0,%1"::"r"(told),"r"(-1.0F)); // told-1
asm volatile("fdiv r26,%0,r26"::"r"(t)); // (told-1)/t
asm volatile("addi r28,r0,0"); // sum=0
```

Código en ensamblador para inicializaciones de variables antes del bucle.

Como se puede apreciar en el código, los registros r20, r21, r22, r26 y r28 están reservados para estas variables, por tanto no se modificaran hasta la finalización del bucle.

```
asm volatile("lwi r23,r20,%0"::"i"(BASE_ADDR_XHAT)); // load xhat[i]
asm volatile("fmul r23,r21,r23"); // xhat*2.0/L
asm volatile("lwi r24,r20,%0"::"i"(BASE_ADDR_W)); // load w[i]
asm volatile("fadd r23,r24,r23");// xhat+w[i]
```

Cálculo de xhat nada mas salir del periférico

Este extracto de código es el que realiza la actualización de xhat cuando el periférico ha terminado su trabajo. El uso de las restricciones "i" permite la

ejecución de instrucciones con operandos inmediatos, de esta forma no se hace uso de más registros que los estrictamente necesarios. Se puede apreciar también que el registro r23 contiene el elemento de xhat final, que va destinado a memoria, pues ya esta calculado.

Desde este punto en adelante, se usa el valor de este registro para todos los cálculos, evitando así escrituras y lecturas a memoria. El resto de operaciones que se hacen en este bucle se hacen de forma análoga a este extracto, instrucciones de tipo aritmético en punto flotante y punto fijo.

El cálculo de temp se hace forma análoga al bucle anterior, por tanto no se muestra aquí ejemplo alguno.

7.10.3. Código en C

En el algoritmo original, tras hacer la minimización L1 de la señal, se ejecuta una ultima vez iMDWT (esto incluye la función bpsconv). Esta parte no se ha portado a ensamblador debido a que la ejecución apenas supera los 10ms, por lo que la ganancia en tiempo no era significativa y se decidió dejarlo en C.

8. Resultados y conclusiones

8.1. Resultados

Antes de exponer los resultados, vamos a presentar un método de cálculo de error para señales ECG, que es el que vamos a usar como referencia durante todo este punto.

El algoritmo se conoce como PRD, y se define como sigue:

$$\mathrm{PRD} = \sqrt{\frac{\sum_{n=1}^{N}\left[y(n)-\hat{y}(n)\right]^2}{\sum_{n=1}^{N}\left[y(n)\right]^2}} * 100\%$$

Donde y(n) es la señal original, a la que se le ha quitado la componente continua (restando la media a cada elemento), e ŷ(n) es la señal reconstruida, también se ha corregido quitando la componente continua (como antes, restando a cada elemento la media del vector). Se establece así, que para valores de entre 0 y 2%, la reconstrucción es perfecta, admisible para valores entre 2 y 9%, y por encima de estos valores requiere estudiar si la señal es buena o no.

El algoritmo implementado tiene un PRD medio de 13.36% en su versión original en MATLAB. A primera vista, esto podría parecer un resultado malo, pero este algoritmo tan sólo es una parte del proceso completo de compresión y descompresión, del cual se obtiene un PRD medio inferior a 2%.

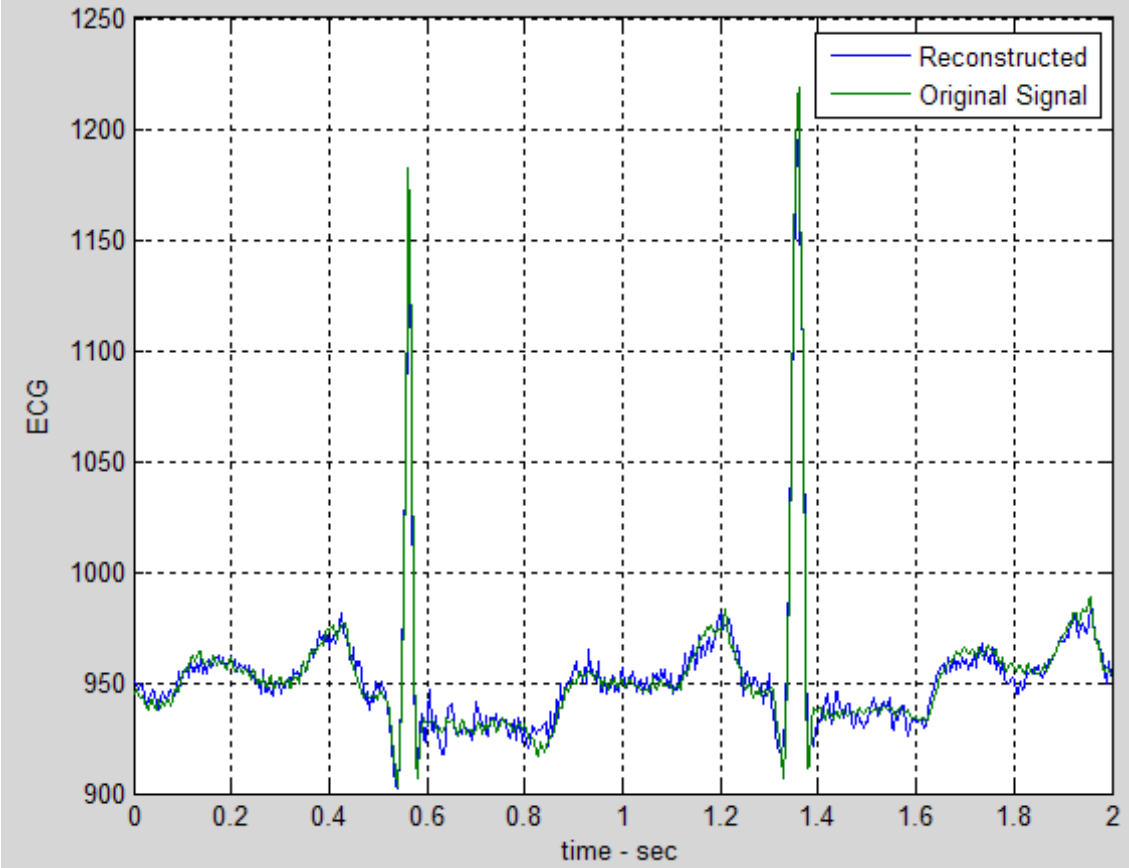
Todos los resultados que se exponen a continuación tienen los mismos parámetros, para hacer una comparación justa. Se emplea para ello un ratio de compresión de un 50%.

Las muestras para probar los algoritmos se obtienen de un fichero binario de datos obtenido de la web www.physionet.org, contiene muestras de ECG de 2 canales, comprimido usando el formato 212, en el que cada elemento de un canal ocupa 12 bits y se guardan 2 elementos de cada canal por cada 24 bits. Todas las señales están revisadas, corregidas y anotadas por médicos cualificados para ello. Cada una de las muestras que aquí se exponen corresponde a 2 segundos de ECG real.

Todos los tiempos mostrados para los algoritmos, si no se indica lo contrario, corresponden a un procesador Intel Pentium 4 2.8GHz, compilados en Visual Studio y corriendo en Windows XP de 32 bits.

8.1.1. Resultados del algoritmo MATLAB

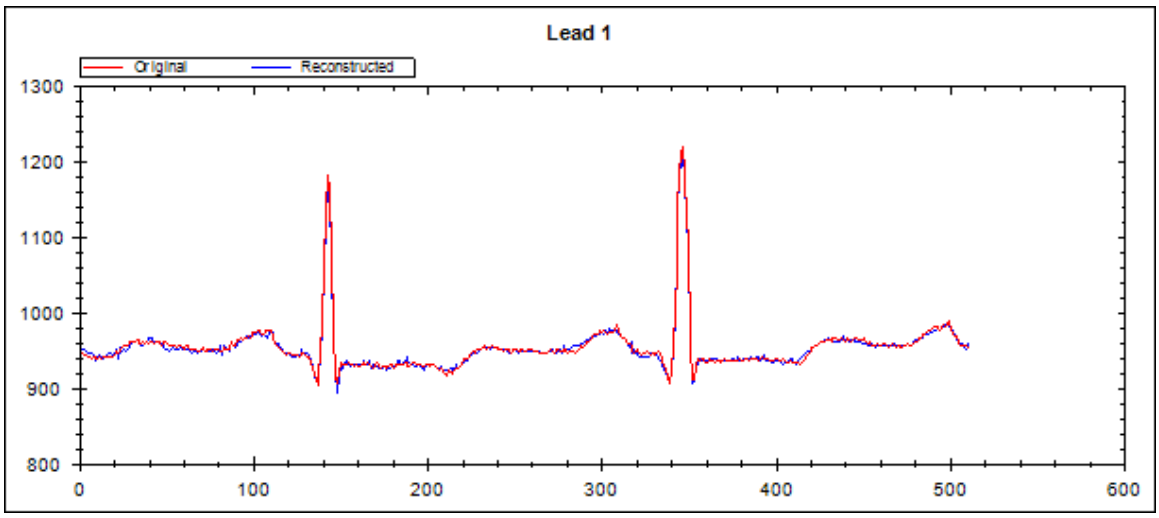
Como se ha reseñado en el punto anterior, el algoritmo original en MATLAB da un PRD medio de un 13.36%, obteniendo unas reconstrucciones aceptables a primera vista:



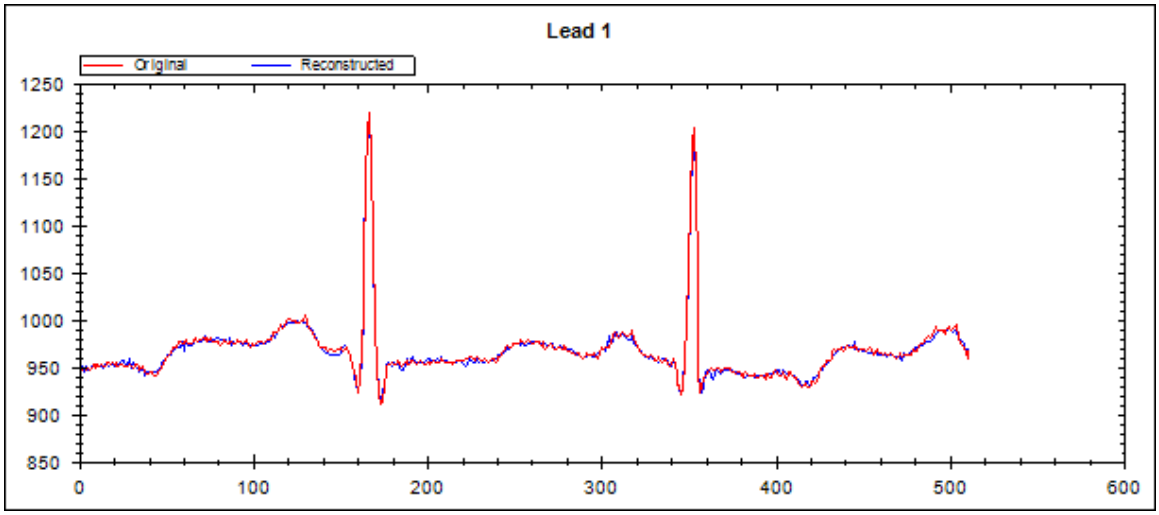
El PRD de esta imagen es 15.33%. Se aprecian un gran número de "artifacts" (picos en la reconstrucción), que son los que dan el PRD tan alto, estos picos se reducen aplicando sucesivos algoritmos de compresión y descompresión de señal, minimizando el error al máximo. El tiempo de ejecución en MATLAB para esta señal en concreto ha sido de 6.91s, y ha empleado 453 iteraciones.

8.1.2. Resultados del algoritmo en C++

Pasamos ahora al algoritmo en C++ que se obtuvo del código original en MATLAB. Pasando el algoritmo por 300 muestras, se obtiene un PRD medio de 12.1%, con un tiempo medio de 6.2s por cada muestra. Esta implementación obtiene unos resultados muy buenos debido al uso de variables en punto flotante con doble precisión.

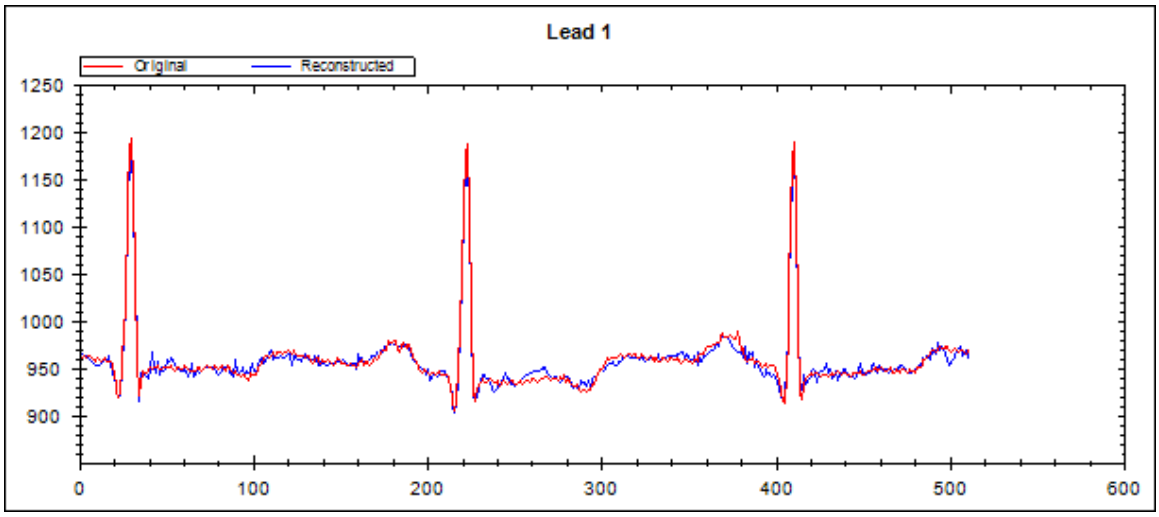


Muestra numero 0 de sel100.dat, 484 iteraciones, 6.11s



Muestra numero 6 de sel100.dat, 476 iteraciones, 6.11s

Las imágenes muestran señales con un PRD de 9.92% y 9.66% respectivamente. Se puede apreciar que las reconstrucciones con un PRD inferior a 10% son especialmente buenas, la diferencia con la señal original es prácticamente inapreciable.



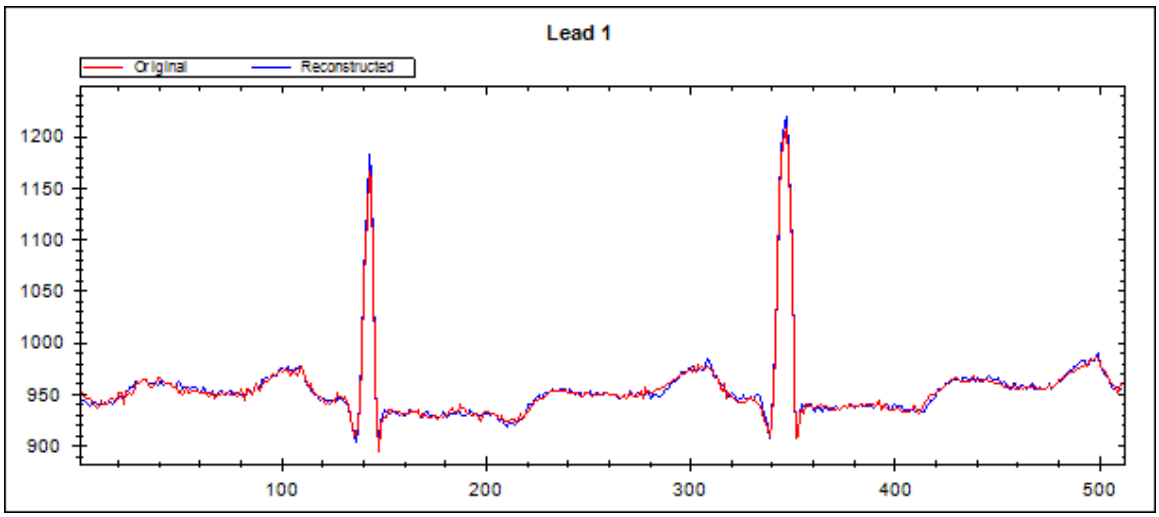
Muestra numero 16 de sel100.dat, 533 iteraciones, 6.45s

Esta imagen tiene un PRD de 15.75%, se puede apreciar la degradación de la señal, pero aun conserva cierta fidelidad a la original.

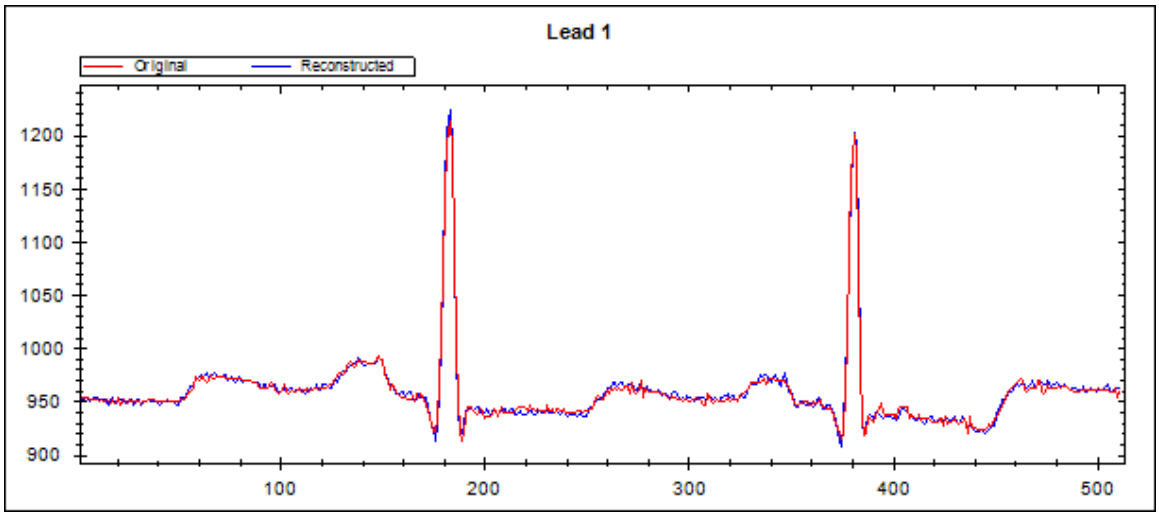
La disparidad en los resultados se debe a la matriz de sensing. Al ser esta aleatoria, y tener únicamente 12 elementos no nulos, estos índices pueden ser los ideales, y aproximar bastante bien una señal tomando solo 12 elementos representativos, pero pueden ser completamente inservibles para otra, aunque generalmente los resultados son muy buenos para cualquier señal.

8.1.3. Resultados código optimizado C

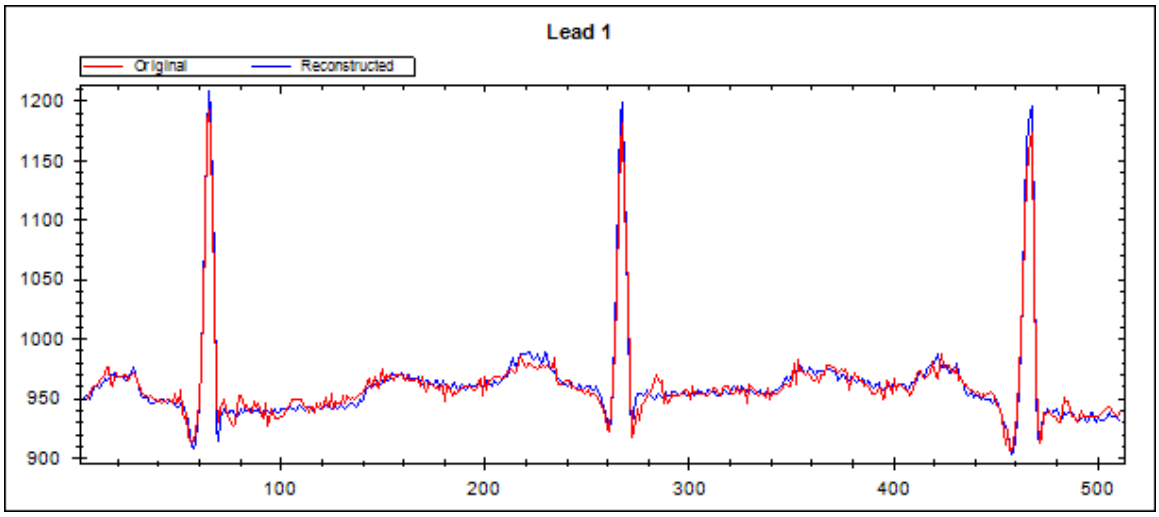
En esta implementación se pierde algo de PRD debido a la utilización de números en punto flotante de precisión simple, pero se gana mucha velocidad. Como antes, pasando el algoritmo por 300 muestras se obtiene un PRD medio de 13.2%, un número medio de iteraciones de 527, y un tiempo medio por muestra de 1.5s. Pasamos a examinar algunas muestras:



Muestra numero 0 de sel100.dat, 486 iteraciones, PRD=9.92%



Muestra numero 200 de sel100.dat, 455 iteraciones, PRD=10.36%



Muestra numero 61 de sel100.dat, 510 iteraciones, PRD=17.91%

Se puede observar que los resultados son prácticamente iguales que los de la versión original en C++. La perdida de precisión es mínima, pero la ganancia en velocidad es muy alta, además de utilizar la mitad de memoria, gracias a la precisión simple en lugar de la doble.

8.1.4. Resultados hardware

Pasamos ahora a revisar los resultados de la implementación final en la placa, con el periférico en hardware y el Microblaze con el código optimizado en ensamblador. Para el análisis de las muestras se ha obviado el tiempo empleado en el envío y la recepción de los datos por el puerto serie, pues no es tiempo de ejecución del algoritmo.

Antes de exponer los resultados de la implementación hardware, se hace notar que el sistema va más despacio de lo que podría por una limitación de la herramienta de síntesis. El sistema esta preparado para funcionar hasta 220MHz, pero debido a problemas y limitaciones de síntesis de Xilinx y de la propia FPGA, se tuvo que bajar la velocidad a 125MHz para asegurar un funcionamiento correcto y consistente. Por esta razón, los tiempos de

ejecución no son todo lo buenos que deberían, ya que con la velocidad máxima admitida, estarían entorno a un 60% de los actuales.

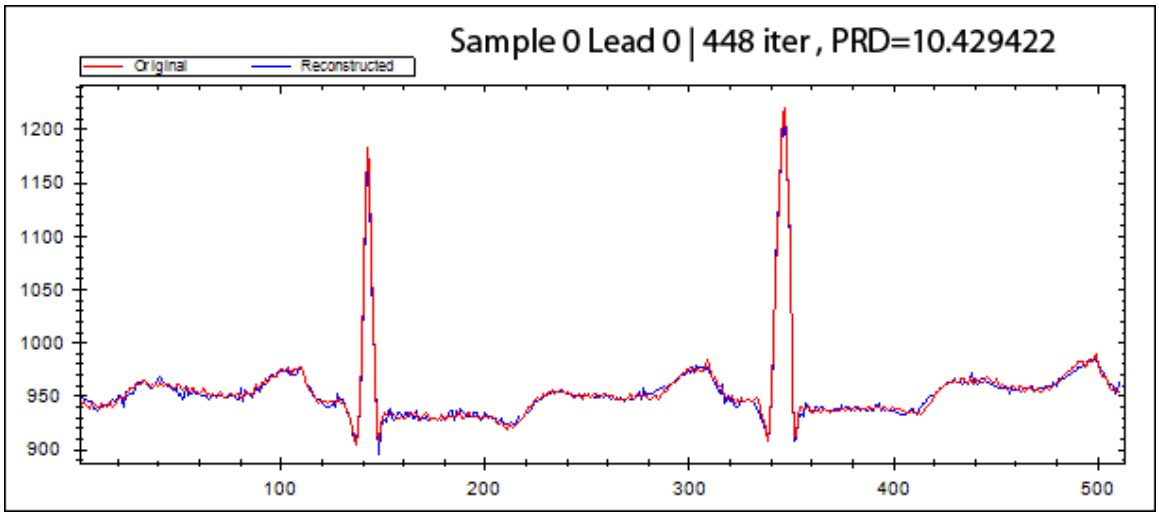
Debido a que la mayoría de los cálculos en flotante no tienen ningún control software, se hacen algunas iteraciones menos en el hardware, por defectos en la precisión que se suplían mediante software. Por esta pérdida de precisión que se va acumulando iteración tras iteración, el error final es ligeramente más alto que en la versión en C, teniendo un PRD medio de 16%. La calidad de las reconstrucciones, visualmente, sigue siendo buena, aunque el PRD sea un poco alto.

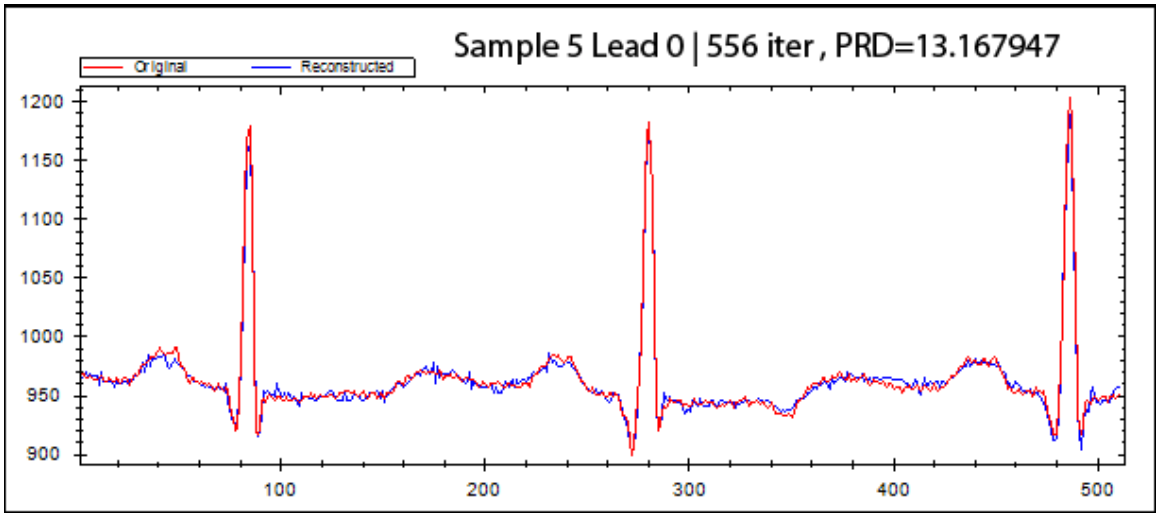
```
Each sample counts as 0.0001 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 95.00      0.58     0.58        1   380.30   380.30  solve_L1
  3.32      0.59     0.01                             memcpy
  1.62      0.60     0.01        9     0.72     0.72  bpsconv_daub
  0.02      0.60     0.00        1     0.10   386.90  main
  0.02      0.60     0.00                             memset
  0.00      0.60     0.00        1     0.00     6.50  iMDWT
```

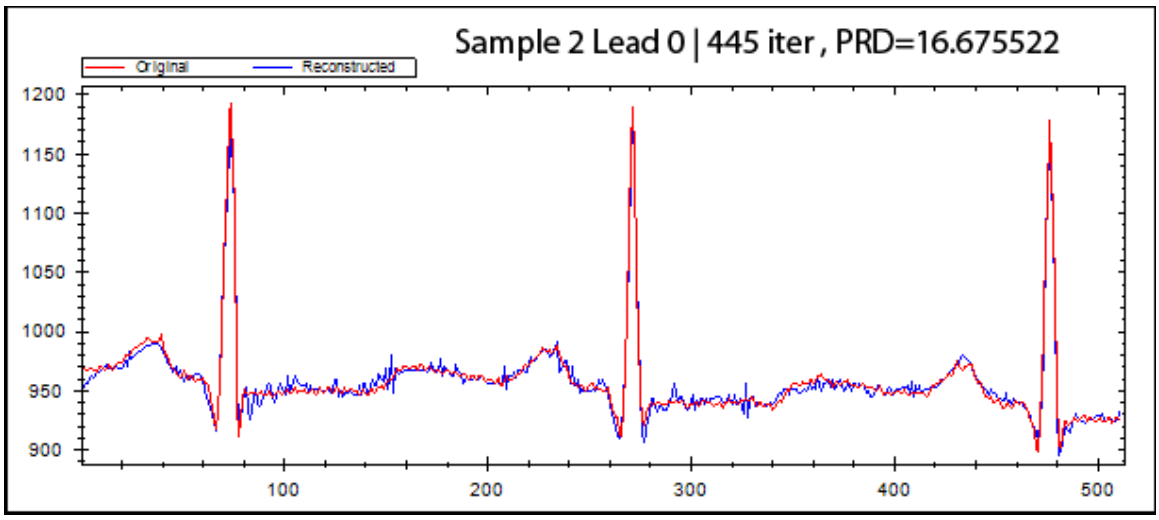
Ejemplo de ejecución para una muestra aleatoria

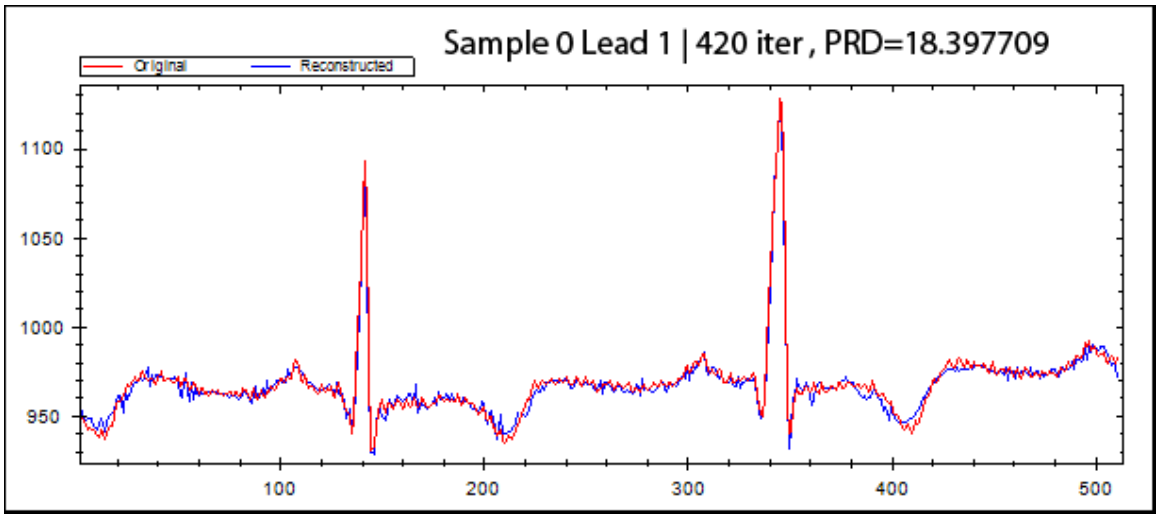
En cuanto a tiempo de ejecución, la media está en 0.7s, que es mejor de lo esperado debido a las primitivas características del hardware empleado. Con esto se cumple la premisa inicial de hacer la descompresión completa en menos de 2 segundos, que es la duración de cada muestra, empleando para ello un hardware muy simple, con muy poca potencia en cálculo flotante y un reloj de tan solo 125MHz.

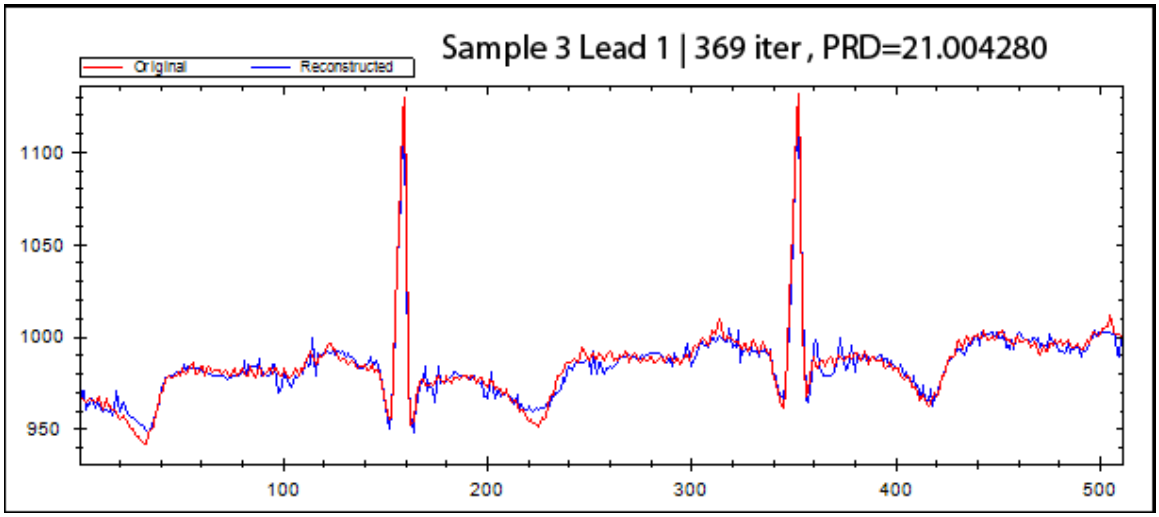
Se muestran a continuación graficas en representación de los mejores resultados (PRD<13%), los medios (PRD=13-19%) y los resultados peores (PRD>19%):

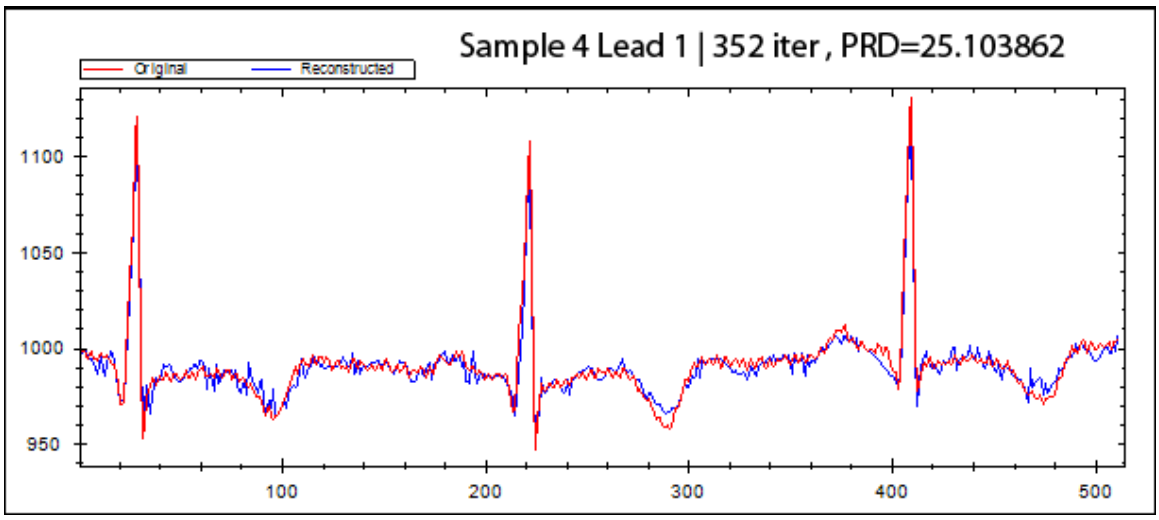



Sample 5 Lead 0 | 556 iter , PRD=13.167947
Original
Reconstructed
1200
1150
1100
1050
1000
950
900
100
200
300
400
500


Sample 2 Lead 0 | 445 iter , PRD=16.675522
Original
Reconstructed
1200
1150
1100
1050
1000
950
900
100
200
300
400
500


Sample 0 Lead 1 | 420 iter , PRD=18.397709
Original
Reconstructed
1100
1050
1000
950
100
200
300
400
500





8.2. Conclusiones

Partiendo de un algoritmo muy complejo, en tiempo, espacio y potencia de calculo necesaria, que era lento en un sistema con arquitectura de 32 bits y un procesador rapido y con un pipeline especial para calculo flotante, la idea de acelerarlo, disponiendo de elementos muy simples, lentos y rudimentarios, se planteaba difícil.

Se ha conseguido implementar el algoritmo, en un tiempo muy por debajo de lo esperado, mediante herramientas que no estan especializadas para calculo en punto flotante, optimizando ademas el espacio utilizado.

El procesador utilizado tan solo dispone de una unidad de punto flotante no segmentada, lenta y que ademas bloquea el procesador hasta que finaliza. Ante esta situación, se opto por utilizar cores creados por xilinx, que hacen uso de los DSP que posee la placa, para acelerar los calculos en punto flotante, obteniendo asi retardos de unos pocos ciclos (4 como maximo) y que ademas admite velocidades maximas de 220MHz. El sistema completo, incluyendo todo el hardware especializado, el procesador, la memoria y todos los modulos necesarios para gestionar la comunicación por serie, apenas ocupa un 30% de una FPGA Virtex 5 VLX110T.

La mejora en tiempo ademas, supera las premisas iniciales, al estar, de media, por debajo de 1 segundo, cuando inicialmente el tiempo superaba los 2 minutos. El speedup medio alcanzado es de 162, y contando con la baja ocupación del sistema, se podria paralelizar para hacer el doble de trabajo en el mismo tiempo.

Como limitaciones del sistema, nos hemos encontrado con un problema en la herramienta de síntesis de Xilinx, asi como la propia FPGA, como se ha mencionado antes, por el cual no hemos conseguido elevar la frecuencia del sistema a la maxima admitida, la cual bajaria el tiempo de ejecución medio por debajo de medio segundo, siendo el speedup aun mayor, en estas condiciones ideales. El sistema funciona a un 40% de su rendimiento ideal, pero aun se obtienen tiempos de respuesta mejores de lo esperado.

9. Bibliografía

> http://www.xilinx.com/support/documentation/index.htm

> http://www.xilinx.com/tools/embedded.htm

> http://www.xilinx.com/tools/microblaze.htm

> http://www.xilinx.com/univ/xupv5-lx110t.htm

> http://www.mathworks.com/help/techdoc/

> http://msdn.microsoft.com/en-us/library/default.aspx

> http://model.com/content/modelsim-support

> A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems. **Amir Beck and Marc Teboulle**

> An introduction to compressive sampling. **Emmanuel J. Candès and Michael B. Wakin**

10. Licencia

Los autores del proyecto "Diseño y optimización para ultra-bajo consumo de arquitecturas y aplicaciones de procesado de señal para las redes de sensores inalámbricas" autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Firmado:

Pablo Acevedo Luis Alfonso González Jorge Guirado